PARALLEL ALGORITHMS FOR LOOK-UP TABLE (LUT) INVERSE HALFTONING

BY

UMAIR FAROOQ SIDDIQI

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

SEPTEMBER 2007

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **UMAIR FAROOQ SIDDIQI** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING DEPARTMENT**.

Thesis Committee

Dr. Sadiq M. Sait (Adviser)

Dr. Alaaeldin Amin (Member)

Dr. Aiman H. El-Maleh (Member)

Dr. Adnan Gutub (Member)

Dr. Abdelhafid Bouhraoua (Member)

Dr. Adnan Gutub
Department Chairman

Dr. Mohammad Al-Homoud
Dean of Graduate Studies (A)

10/9/07
Date

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

ACKNOWLEDGMENTS

This thesis would have been taken far more time without the help and support of others. Foremost, I would like to thank my research adviser Professor Sadiq M. Sait for his guidance and support throughout my MS completion. His attention, advices and consistent focus kept the path lit amidst a forest of distraction. I would also like to thank Professor Alaaeldin Amin, Professor Aiman H. El-Maleh, Professor Adnan Gutub and Professor Abdelhafid Bouhraoua for kindly reviewing this effort. I would like to thank Professor Dr. Mayez Abdullah Al-Mouhamed for his time and devotion to help me excel my skills. I also would like to acknowledge Dr. Aamir Alam Farooqui for his kind advises to improve the quality of work. I would like to acknowledge my all teachers at King Fahd University of Petroleum & Minerals, Dhahran for their guidance and help.

I would like to thank Mr. Hafeez, Mr. Khurshid and Mr. Yousuf for their help in administration related issues. I also would like to thank people at Information Technology Center (ITC) for their support. I would also like to thank my friends at computer engineering department for their encouragement and help with arabic translation of the abstract.

I like to thank my parents, brothers and sisters and other relatives for their prayers and continual encouragement for me to pursue studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME: Umair Farooq Siddiqi

TITLE OF STUDY: Parallel Algorithms for Look-Up Table (LUT) Inverse Halftoning

MAJOR FIELD: Computer Engineering Department

DATE OF DEGREE: September 2007

The Look-Up Table (LUT) method for inverse halftoning is fast and computation-free technique employed to obtain good quality images. In this work we propose six algorithms to parallelize the LUT method so that more pixels can be concurrently inverse halftone using minimum additional hardware. The proposed algorithms partition the single LUT of serial LUT method into N smaller Look-Up Tables ($s - LUTs$) such that the total number of contents in all $s - LUTs$ remain equal to the number of contents in the single LUT of serial LUT method. The proposed parallel algorithms have image quality equal to the serial LUT method when gain in clock cycles over the serial method is less and have lesser image quality comparetively to serial LUT method when gain in clock cycles over the serial method is very high. The parallel algorithms can be implemented on $FPGA$ (Field Programmable Gate Arrays) devices with external CAM (Content Addressable Memories) and ROM (Read Only Memories).

ملخص الرسالِة

الاسم: عمير فاروق صديقي

العنوان الرسالة: اللوغاريحات التوازية لقوائم النصاف الاخبار العكية الجاهزة

التحصص: علوم الكومبيوتر هندسة

التاريخ التخريج: سبتمبر 2007

استخدام طريقة الجداول المصفوفية LUT لمعكوس نصف اللون(HalfToning)، التي تتميز بالسرعة والتي لا تحتوي على حسابات رياضية ،التي تستخدم للحصول على صور عالية الدقة في هذه الرسالة نقترح 6 طرق algorithms لإستخدام طريقة LUT على التوازي بحيث ان أكثر من نقطة على الصورة (pixel) تستخدم معكوس نصف اللون inverse halfetoning وذلك باستخدام اقل كمية ممكنة من القطع الإضافية hardware. الطرق المقترحة تجزء وحدة من LUT متسلسل ن قطع اصغر s-LUT بحيث يكون عدد القطع المحتوية الإجمالي مساوي لعدد القطع المحتوية في وحدة LUT الطريقة المقترحة تعطي نفس جودة الصورة عندما تكون نسبة الكسب الناتجة عن نسبة عدد دورات الساعة على الطريقة التسلسلية اقل. وتكون الصورة اقل جودة إ1ذا كان نسبة الكسب الناتجة عن نسبة عدد دورات الساعة على الطريقة التسلسلية عالية جدا. و يمكن تطبيق هذه الطريقة باستخدام FPGAمع CAM (Content Addressable Memories) خارجي ووحدة ذاكرة ROM .

CHAPTER 1

INTRODUCTION

1.1 Halftoning

Halftoning is the rendition of continuous-tone pictures on media on which only two levels can be displayed. It was first employed in the late 19^{th} century when printing machines were used to print images on paper. Halftoning was accomplished by adjusting the size of the dots according to local image intensity [1]. This halftoning is called analog halftoning. With the proliferation of bi-level devices, digital halftoning also gained importance. Some of these bi-level devices are fax machines, printers, and plasma display panels. Halftoning is vastly used in printing newspapers, magazines, etc. The first commercial halftone screen was used in 1866 to print continuous tone images on paper by arranging dots for a specific image. Some of the screen densities used are 50-85 lpi (lines per inch) for newspapers, 100-120 lpi for highly polished papers and for some magazines, 120-150 lpi for color illustrations in magazines and for books printed on coated papers.

Figure 1.1: Illustration of a continuous tone image.

1.1.1 Digital Halftoning

Digital halftoning is the rendition (translation) of continuous-tone pictures on displays that are capable of producing only two levels. The input is an image in which each pixel has more than two levels, e.g. 256 levels, and the result of the halftoning process is an image in which each pixel has only two levels, i.e., 0 or 1. These two images look nearly same to human eye when viewed from a distance. Thus halftoning process exploits the low pass characteristics of the human visual system. A continuous-tone image is shown in Figure 1.1 followed by its halftoned version in Figure 1.2. The performance of different halftoning algorithms is measured by: computational complexity, halftone quality, and their capability to achieve parallel operation. Some algorithms are summarized in Table 1.1. In Table 1.1 ordered dither algorithm is the simplest one and has complete parallelism but has worst halftone quality. On the other hand error diffusion have good image quality but error diffusion is inherently serial. The best quality is achieved by direct binary search algorithm, but it is computationally very intense. The dot diffusion method for digital halftoning has the advantages of

Figure 1.2: Illustration of a halftone image.

Table 1.1: Comparison of different halftoning algorithms.

Method	Complexity	Halftone Quality	Parallelism
Ordered Dither [1]	extremely low	poor	complete
Dot Diffusion [1]	low	poor	substantial
Error Diffusion [1]	low	good	none
Direct Binary Search [1]	high	best	none

pixel-level parallelism unlike the popular error diffusion halftoning method. However, image quality offered by error diffusion is still regarded as superior to most of the other known methods. A Look-Up Table (LUT) method for halftoning is proposed by Mese and Vaidyanathan [1] in recent past. In it pixels from a casual neighborhood (template) and contone value (or gray level value) of the current pixel will be included in the LUT. The LUT method has image quality between error diffusion and Direct Binary Search (DBS). The LUT halftoning is illustrated in Figure 1.3 where the pixel represented by *?* is the pixel whose bi-level value is to be evaluated and the other neighboring pixels in the gray shade form the complete template mentioned above.

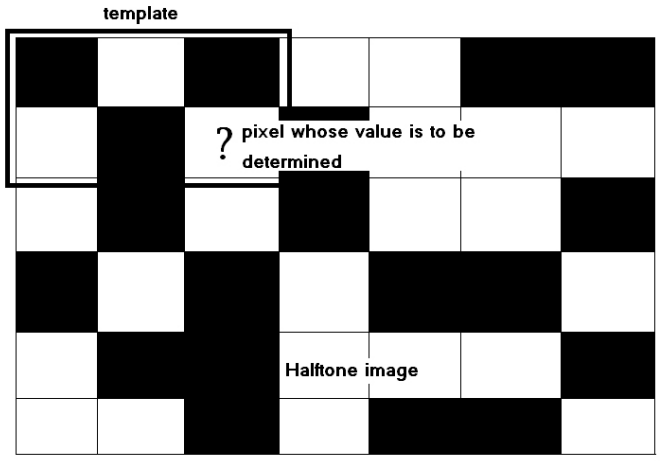

Figure 1.3: Illustration of LUT halftoning.

1.2 Inverse Halftoning

In Inverse halftoning, continuous tone (or gray level) images are reconstructed from their halftone versions. It has wide range of applications. Examples include image compression, printed image processing, scaling, enhancement, etc. In these applications, operations cannot be done on halftone image directly, and inverse halftoning is mandatory.

In recent past Look-Up Table (LUT) methods for inverse halftoning have gained attention because they offer fastest inverse halftoning as compared to the other methods. One LUT method is proposed by Mese and Vaidyanathan [6]. In this method the LUT is obtained from the histogram gathered from a few sample halftone images and corresponding continuous tone images. For each pixel, it looks at pixel's neighborhood (template) and depending upon the distribution of pixels in template a contone value (or gray level value) is assigned from a precomputed LUT. The method is illustrated

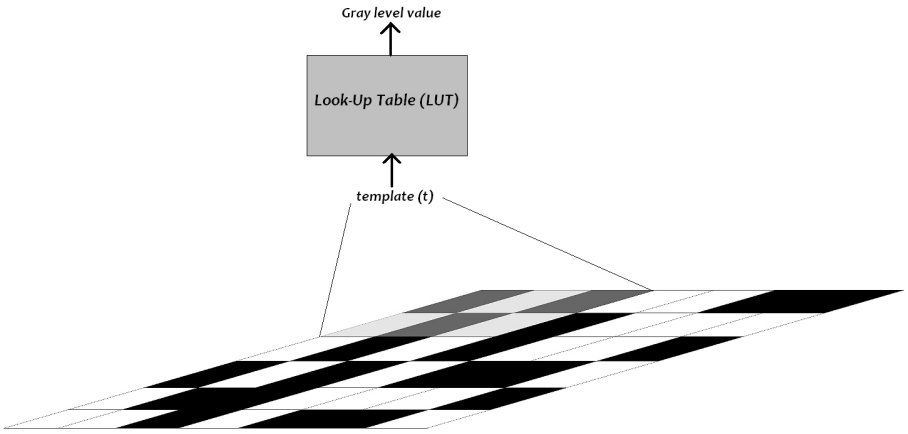

Figure 1.4: Look-Up Table (LUT) inverse halftoning proposed by Mese at al.

in Figure 1.4. It is extremely fast (no filtering is required) and the image quality achieved is comparable to the best methods known for inverse halftoning. It does not depend on the specific properties of the halftoning method, and can be applied to any halftoning method. This method can also be extended to color halftones. A comparison of different LUT inverse halftoning methods is shown in Table 1.2.

Table 1.2: Comparison of different inverse halftoning algorithms.

LUT Method	Computation	LUT Size	Image Quality
Mese at al.[6]	No	medium	good
Cheng at al.[8]	Yes	medium	good
Chung et al.[9]	Yes	large	very good

1.3 Parallelization of LUT Inverse Halftoning

The parallelization of Look-Up Table (LUT) inverse halftoning is an interesting problem. It is important because it can increase the speed of LUT inverse halftoning by concurrent inverse halftoning two or more pixels. The increase in speed benefits real-time image processing of halftone images as performed in scanners and other devices.

The LUT is stored in a single memory unit. A single memory unit has a single address input e.g. a Read Only Memory (ROM) has only one input for address. Therefore only one value can be fetched from a Look-Up Table (LUT) at a time. This limitation makes the LUT methods of inverse halftoning serial. To accomplish parallel LUT inverse halftoning in which two or more pixels are inverse halftoned concurrently we try to fetch more than one value from the Look-Up Table (LUT) at the same time. The solution of parallelization depends on the target platform that will be used to implement the LUT inverse halftoning. One solution that can work with a variety of platforms is to partition the single LUT present in the LUT methods into two or more parts. In this way two or more values can be fetched concurrently from different parts. It also requires that entries to be conurrently fetched must be from distinct partitions of the LUT. The target platforms for this approach are: Field Programmable Gate Arrays (FPGA) in which the memory may consists of separate modules with independent I/O ports, Application Specific Integrated Circuits (ASIC), and Microprocessor with independently accessible memory banks. The parallel LUT inverse halftoning is illustrated in Figure 1.5 in which five templates are fetched and inverse halftoned at the same time.

1.3.1 Introduction to Proposed Algorithms for Parallel LUT Inverse Halftoning

The algorithms proposed to perform parallel Look-Up Table (LUT) inverse halftone operation enhances the serial LUT method [6] such that up-to k templates can be

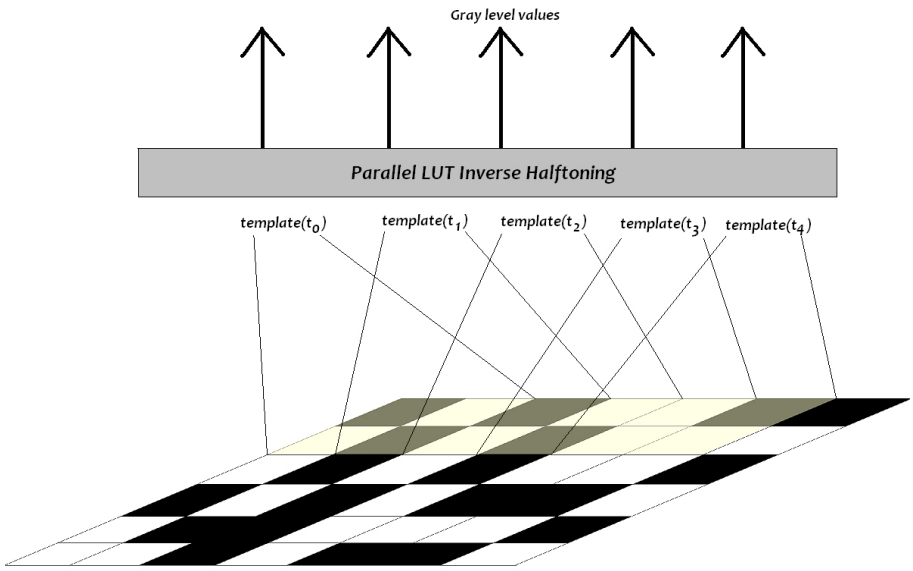

Figure 1.5: Inverse halftoning using parallel LUT method.

simultaneously fetched from the halftone image and obtain their contone values at the same time. It consists of three steps: (1) Defining functions that perform the task to distinguish two or more templates from each other, (2) Generation of N smaller Look-Up Tables ($s - LUTs$) through partitioning the single LUT of serial LUT using the function described in step 1. Total entires in N $s - LUTs$ remains equal to the entries in the single LUT of serial LUT method, and (3) Algorithms to perform parallel LUT inverse halftoning with minimum computation using N $s - LUTs$.

In the proposed algorithms, N smaller Look-Up Tables ($s - LUTs$) are numbered from 0 to $N - 1$ and the function defined are named as: "Relative XOR Change" (RXC), "XOR with Mean" (XM), "Only Addition" (OA), and "Mod Only" (MO). An example of partitioning of single LUT into 8 $s - LUTs$ is shown in Figure 1.6, in which the $s - LUTs$ are generated directly from the training set then first generating the single LUT and peform partitioning. The illustrations of parallel LUT inverse

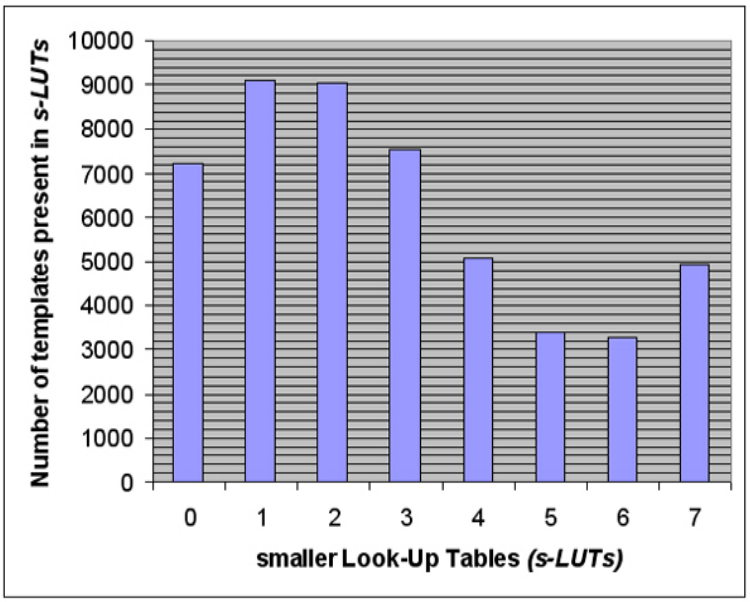

Figure 1.6: Graph showing partitioning of the contents of single LUT into 8 $s-LUTs$ (smaller Look-Up Tables).

halftoning using the proposed functions is shown in Figure 1.7, Figure 1.8, and Figure 1.9.

The proposed algorithms also contain algorithms to generate smaller Look-Up Tables ($s-LUT$) that must be generated before inverse halftone operation is performed. These $s-LUT$ generation algorithms consists of RXC, XM, MO or OA functions and same function should be used in $s-LUT$ generation and in inverse halftone operation.

The serial LUT method is completely computation free. However, the proposed parallel LUT inverse halftoning have simple, fast and pipelined computational circuit before look-up tables. A FPGA implementation of the proposed parallel LUT inverse halftoning is illustrated in Figure 1.11, where it is assumed that the complete computational circuit is implemented in the FPGA and $s-LUT$ are stored in external memories.

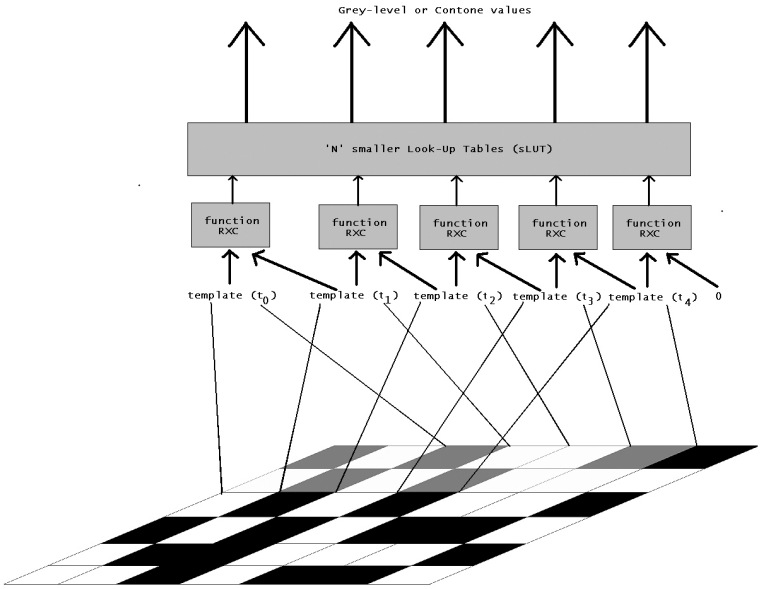

Figure 1.7: Parallel LUT inverse halftoning with function RXC (Relative XOR Change).

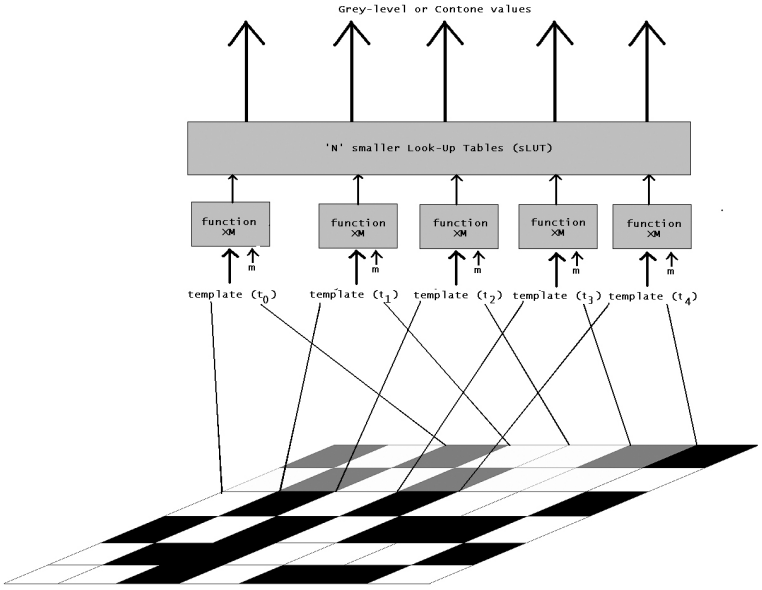

Figure 1.8: Parallel LUT inverse halftoning with function XM (Xor with Mean).

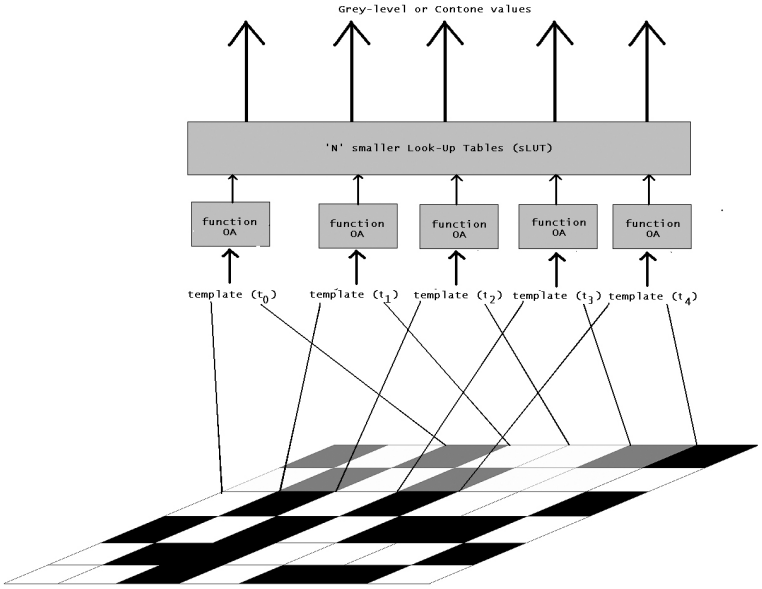

Figure 1.9: Parallel LUT inverse halftoning with function OA (Only Addition).

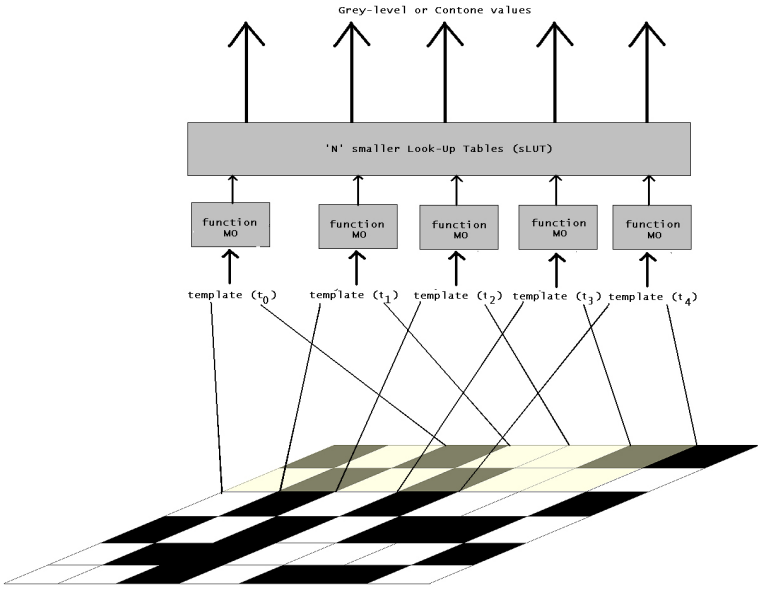

Figure 1.10: Parallel LUT inverse halftoning with function MO (Mod Only).

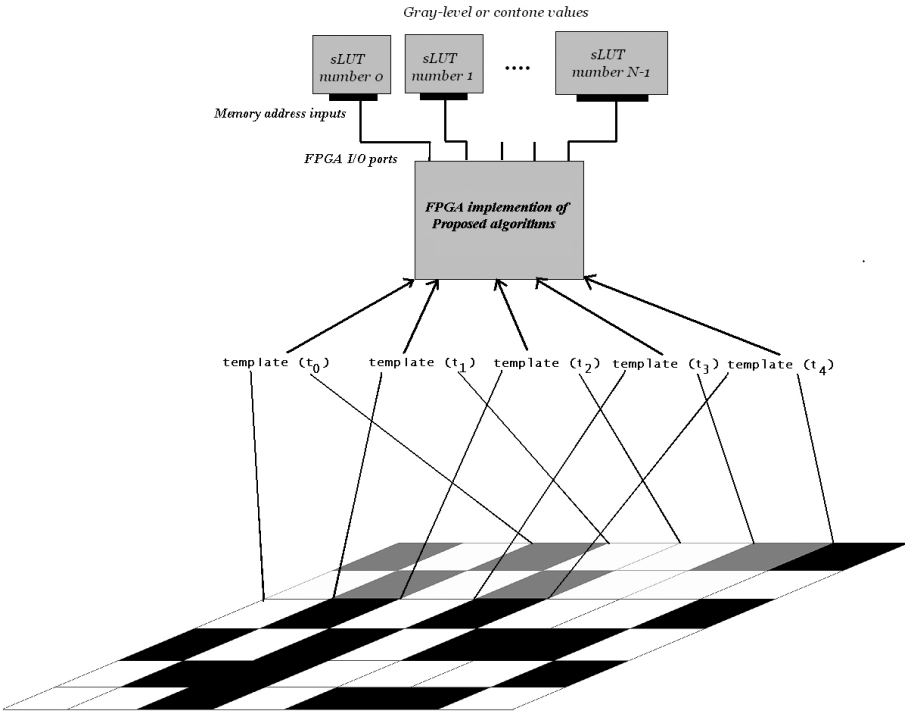

Figure 1.11: Illustration of hardware implementation of the proposed parallel LUT method.

1.4 Applications of Inverse Halftoning

The inverse halftoning is used in Scanner Model Based Method (SMB) that is typically used in scanners to recover printed images as continuous-tone images.

1.4.1 Inverse Halftoning in Scanner Model Based Method (SMB)

The SMB [7] consists of five steps as shown in Figure 1.12. The first step is the Scanning Process that consists of two parts. First part performs the transformation from printed image dot *(D(i,j))* to the brightness received by the scanner sensor. It is assumed that the brightness received by a sensor from any dot on the paper de-

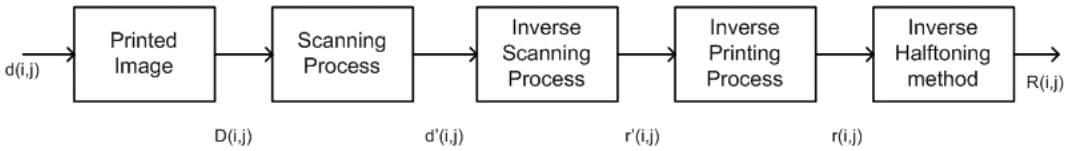

Figure 1.12: The steps in scanning process using inverse halftoning.

pends only on the relative distance generated by all dots and the sensor, and that the brightness generated by all dots can be linearly-superimposed. Under these assumptions, this transformation is represented by a linear filter with impulse response *(h(i,j))* which behaves like a low pass filter. Second part represents the transformation from the brightness that the sensor received to its output value. It behaves like a clamping function that limit the scanner outputs to a finite range. The second step in SMB is Inverse Scanning Process which consists of inverse clamping operation and the inverse scanner linear filter. The inverse scanner filter is found by finding the inverse model of the Scanner Process. The third step in SMB is Inverse Printing Process that maps each pixel to either 0 or 255 because the Inverse Scanner Process has more than two levels. The final step in SMB is the inverse halftoning. In this step any inverse halftoning algorithm can be used to recover original-like continuous-tone image from the halftone image or image obtained after Inverse Printing Process.

1.5 Contents of Thesis

This thesis is organized into chapters and each chapter shows a distinct information that is required to describe the work conducted during the thesis research. In Chapter 2 we showed that inverse halftone algorithms can be computation based or Look-Up Table (LUT) based. The computation based algorithms have good image quality, less

memory requirements but generally require complex computation in order to perform inverse halftone operation. The LUT inverse halftoning methods are fast, comparable image quality, very less computational and have high memory requirements. A LUT method proposed by Mese and Vaidyanathan [6] is completely computation free. The chapter then describes the LUT method in detail. This chapter in the end concludes that Mese et al. [6] LUT method is selected to be enhanced in order to perform parallel LUT inverse halftone operation. Because it is the fastest and only method that is completely computation free. Chapter 3 presents four new algorithms that are proposed to perform parallel LUT inverse halftoning. The parallel LUT inverse halftoning consists of three steps: (1) Definition of functions to help in sending templates to distinct smaller Look-Up Tables ($s-LUT$), (2) Generation of $s-LUT$ using training set comprising of halftone images and corresponding continuous-tone images, and (3) Parallel LUT inverse halftone operation. In Chapter 4 we show simulation of the proposed algorithms using a toolbox. The toolbox is developed in Java programming language and contains implementations of serial and proposed parallel LUT methods. The toolbox gives algorithms performance in terms of image quality, clock cycles consumed in inverse halftone operation and show resulting images obtained after inverse halftone operation. The toolbox employs a graphical interface for setting algorithm parameters like N i.e., number of $s-LUTs$, and k i.e., number of concurrently fetched templates. It also accepts images to generate LUT and perform inverse halftone operation. A screen shot of the toolbox is shown in Figure 1.13. Chapter 4 presents hardware modeling of the proposed algorithms using VHDL and when target

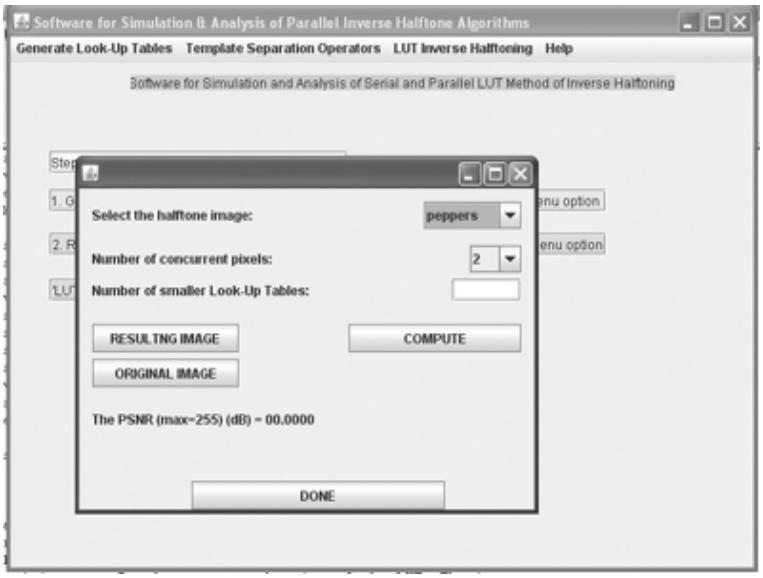

Figure 1.13: Graphical Interface of the Simulation Toolbox.

is Xilinx Spartan 3E FPGA (Field Programmable Gate Arrays) that has $100K$ gates. The model is synthesizable and design statistics are obtained from Xilinx ISE tools. A place and route of net-list of a circuit implementing one of the proposed algorithms is shown in Figure 1.14. The circuit only consists of computational part. In Chapter 6 we show the conclusions we can draw from the thesis research. We also discuss the usefulness of the thesis research.

1.6 Summary

In this chapter it is shown that halftoning is the process of converting continuous-tone images into bi-level images. It is used in printing newspapers, magazines, books, etc. Digital halftoning is the rendition of continuous-tone pictures on displays that are capable of producing only two levels. Some devices using digital halftoning are printers, plasma display panels etc. Many different halftoning algorithms exist that

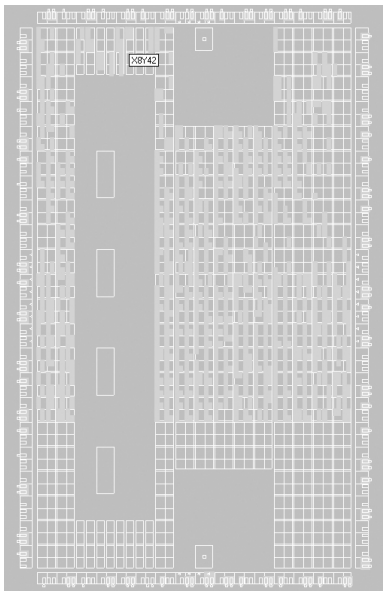

Figure 1.14: Illustration of floorplan showing mapping of computation in the proposed algorithms to the target FPGA.

offer different image qualities, memory requirements and computational complexities. Inverse halftoning is the reconstruction of continuous-tone images from their halftone versions. Algorithms that perform inverse halftone operation can be computation based or Look-Up Table (LUT) based. In LUT methods no or very less computation is required to perform inverse halftone operation. The memory requirements are however very large. All LUT methods are serial because they use a single LUT and one cannot fetch more than one value from a LUT at a time. The LUT methods can be parallelized to further increase the speed of inverse halftoning. It benefits devices such as scanners to perform fast real time inverse halftoning. Four new algorithms are proposed to perform parallel LUT inverse halftoning. They consist of: (1) N look-up tables called smaller Look-Up Tables ($s - LUT$) in place of a single LUT and N $s - LUTs$ contain total entries equal to the entries in the single LUT of serial LUT methods, (2) More than one pixels can be fetched and inverse halftoned concurrently

using $N\ s - LUTs$. Inverse halftoning is used in scanners using Scanner Model Based (SMB) method in which inverse halftone operation is performed at five stages to recover continuous-tone images from printed images.

CHAPTER 2

LITERATURE SURVEY

Since there can be more than one continuous tone image giving rise to a particular halftone image, there is no unique inverse halftone of a given halftoned image. Thus, extra properties of images are needed in order to do inverse halftoning. The basic assumption in all inverse halftoning algorithms is that "natural" images have "mostly lowpass" characteristics. Simple low pass filtering can remove most of the noise injected by halftoning algorithm but it also removes edge information [6]. Besides lowpass filtering more sophisticated approaches exist. The method of Projection Onto Convex Sets ($POCS$) has been used for inverse halftoning of ordered dithered images by Analoui and Allebach [12] and for error diffused images by Hein and Zakhor [13]. Fan [14] has used a method logical filtering on order dithered images. Iterative filtering has been used by Wong [15] to inverse halftone error diffused images. The method of over-complete wavelet expansions has been used by Xiong [16] to produce inverse halftones with good quality for error diffused images. It is accomplished by separating the halftoning noise from original images through edge detection. Another

method for inverse halftoning of error diffused images was introduced by Kite et al. [11]. This method is not only fast but also yields images of very good quality. The Look-Up Table (LUT) methods for inverse halftoning are fastest and of low computation among all other methods. Three LUT methods are proposed by Mese et al. [6], Chang et al. [8], and Chung et al. [9]. All previous algorithms for inverse halftoning can be divided into two categories:

1. Computation based inverse halftoning algorithms

2. Look-Up Table (LUT) based inverse halftoning algorithms

2.1 Computation based Inverse Halftoning Algorithms

In this section we show several computation based algorithms for inverse halftoning.

2.1.1 Inverse Halftoning using Fourier Deconvolution

Deconvolution can be used to perform inverse halftoning [20, 21, 22]. It consists of two steps. In the first step the input halftone image represented by $y(n_1, n_2)$, where n_1 is the number of rows and n_2 is the number of columns, goes through *Operator Inversion*. In this step the halftone image is converted into a gray level image with color noise present in it. The next step is *Fourier Domain Shrinkage* in which the color noise component is attenuated to obtain the output gray level image. The images obtained after Fourier Domain Shrinkage do not have sharp edges because of excessive noise in

the image or due to distortions such as blurring or ringing in the image. The resulting images have large MSE (Mean Square Error).

2.1.2 Inverse Halftoning by Wavelets

The wavelet transform provides an economical representation as compared to other transforms [22]. It also consists of two steps. The first step *Operator Inversion* converts the input halftone image $y(n_1, n_2)$ to a gray level image with noise component present in it. The next step *Wavelet Domain Shrinkage* performs attenuation of noise component in the wavelet domain. This shrinkage is called Wavelet Inverse Halftoning via Deconvolution ($WInHD$) and it first applies DWT (Discrete Wavelet Transform) on the input image and then the noise components are attenuated. This shrinkage also exploits the wavelet representation to perform economical representation.

2.1.3 Fast Inverse Halftoning

This method is proposed by Kite et al. [11]. A low-pass filter imposes a fixed relationship between the increase in gray scale resolution and decrease in spatial resolution. By spatially varying this trade off large improvement in inverse halftone quality can be obtained. In this algorithm inverse halftone operation is performed by applying appropriately chosen smoothing filters at each pixel in the halftone image. These filters perform spatially varying linear filtering. The method has 30-226 integer additions, seven integer multiplications, 34 floating-point additions, 22 floating-point multiplications, 6 floating-point divisions, and 303 increment operations. Memory requirement

is $7(c+6)$ bytes where c is the number of image columns. The image quality is better than many other algorithms.

2.2 Look-Up Table (LUT) based Inverse Halftoning Algorithms

LUT inverse halftoning was first introduced by Netravali and Bowen [4] but it requires some information to be known that is not always available with halftone images. Subsequently Ting and Riskin [5] proposed another LUT method but it did not yield good image quality. Recently three methods for LUT inverse halftoning are proposed by Mese at al. [6], Cheng et al. [8] and Chung at al. [9] These methods have good image quality and have no or very less computation. In the following we review these three algorithms:

2.2.1 Look-Up Table (LUT) Method by Mese and Vaidyanathan

The Look-Up Table (LUT) method proposed by Mese and Vaidyanathan [6] define a template *(t)* that consists of the pixel to be inverse halftoned and its neighboring pixels. It uses three types of templates namely: *16opts*, *19opts*, and *Rect*. The *16opts* consists of 17-pixels, *19opts* consists of 20-pixels and *Rect* consists of 21-pixels. The template *19opts* is shown in Figure 2.1, in which template number 0 is the one to be inverse halfoned and other pixels are in its neighborhood. The numbering of pixels

in the template also specify their bit positions for storage in registers. The templates are fetched from halftone image following raster-scan style i.e., from left to right in a row and travel rows from top to bottom. One template is fetched and inverse halftoned before next template is fetched from the halftone image. Prior to start inverse halftone operation the Look-Up Table (LUT) has to be developed. The LUT is developed by building a training set that consists of continuous tone images and their corresponding halftone images. The halftone images are obtained from continuous tone images by applying any halftoning algorithm like Floyd and Steinberg [10] on them. The templates are fetched from halftone images and stored in the LUT, the corresponding contone values are also fetched from halfone images along-side and stored in the LUT at locations adjacent to their templates. When templates in all training set images are fetched and stored in the LUT the training is completed. If a template occurs more than once in the training set then its corresponding contone value is the mean of all contone values that occur in the training set for that template. The inverse halftone operation is performed in this way, that a template say t is fetched from the halftone image and it goes to the LUT, where the contone value corresponding to that template should already be present and template then returns a contone value from the LUT. Now we fetch the next template and continue this process until we inverse halftoned the complete image. In case, when the template t cannot find its contone (or gray level) value in the LUT, the methods best Linear Estimator or Linear Filtering [6] are applied to the template to find its contone value. However, when same halftone algorithm is used in halftone images as well as in building LUT i.e.,in

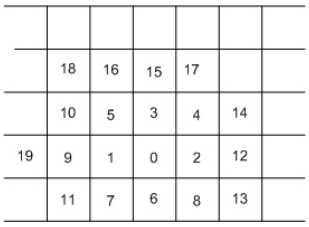

Figure 2.1: Template "19opts".

training set halftone images then all fetched templates always find matching templates in the LUT. This method can also be extended to color inverse halftoning in which three separate LUT are developed for planes R, G, and B using the same method as described. The performance of the LUT method was illustrated [6] by building the LUT from a training set of 30 images and then inverse halftoning test images *lena* and *mandrill* using the LUT method and another method called $fastiht2$ [11]. The $fastiht2$ is also a recent algorithm that has good image quality. Experiments have shown that LUT method has 7.5% and 3.1% lower $PSNR$ (Peak Signal to Noise Ratio) as compared to the $fastiht2$ method. The images quality is still considered to be good. Two images for visual quality comparison are shown in Figure 2.2 and Figure 2.3.

The LUT method is computation free and consists of a single Look-Up Table (LUT) of size around $476K$ entries when template type is *19opts*. The VLSI (Very Large Scale Integrated Circuits) implementation consists of a CAM (Content Addressable Memory) and a ROM (Read Only Memory). The CAM stores all templates that are obtained for the LUT through training set and it returns the addresses of locations in ROM where their corresponding contone values are stored. Therefore, ROM stores the contone values of templates. The inverse halftone operation is performed in this

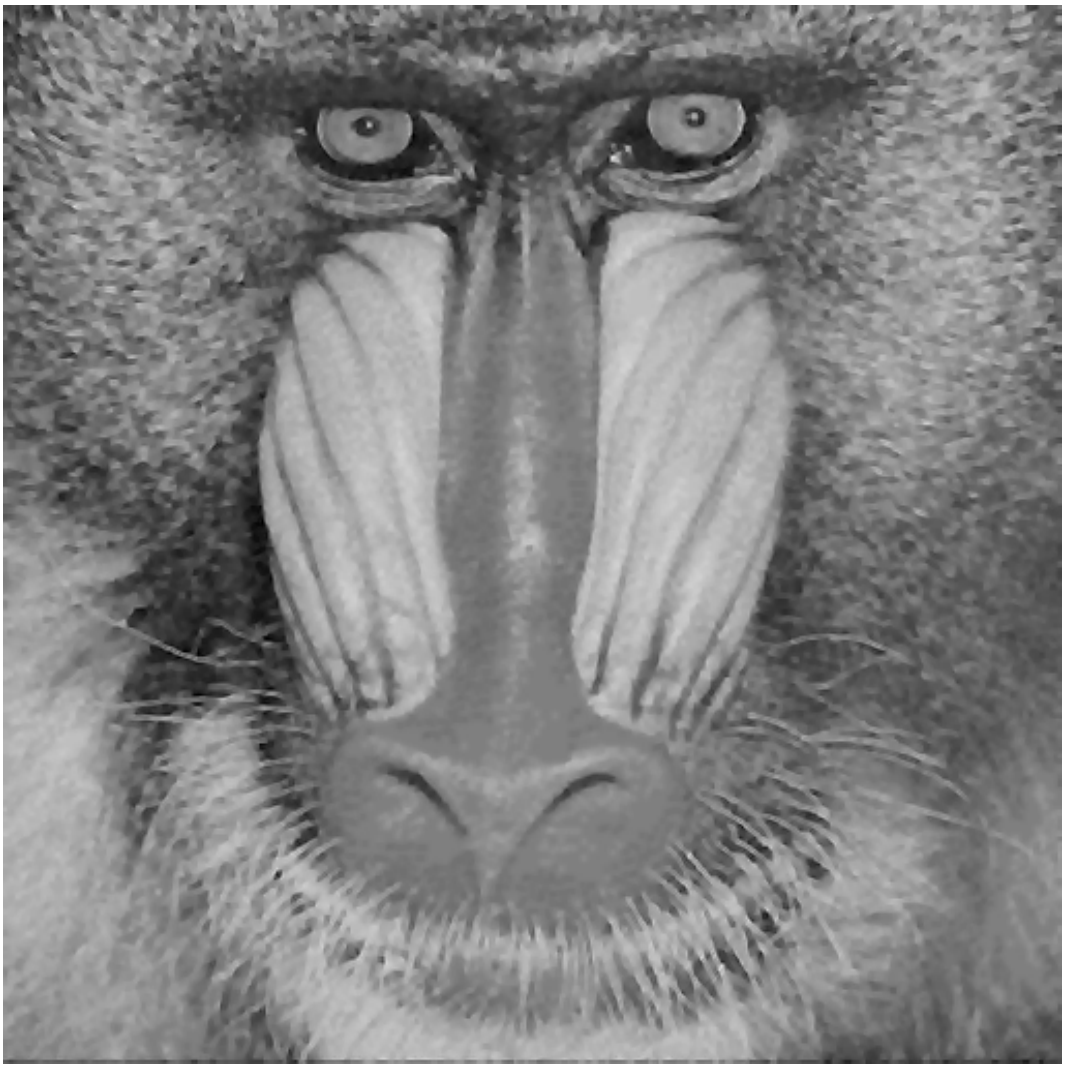

Figure 2.2: Image obtained by using Fast Inverse Halftoning (fastiht2) algorithm.

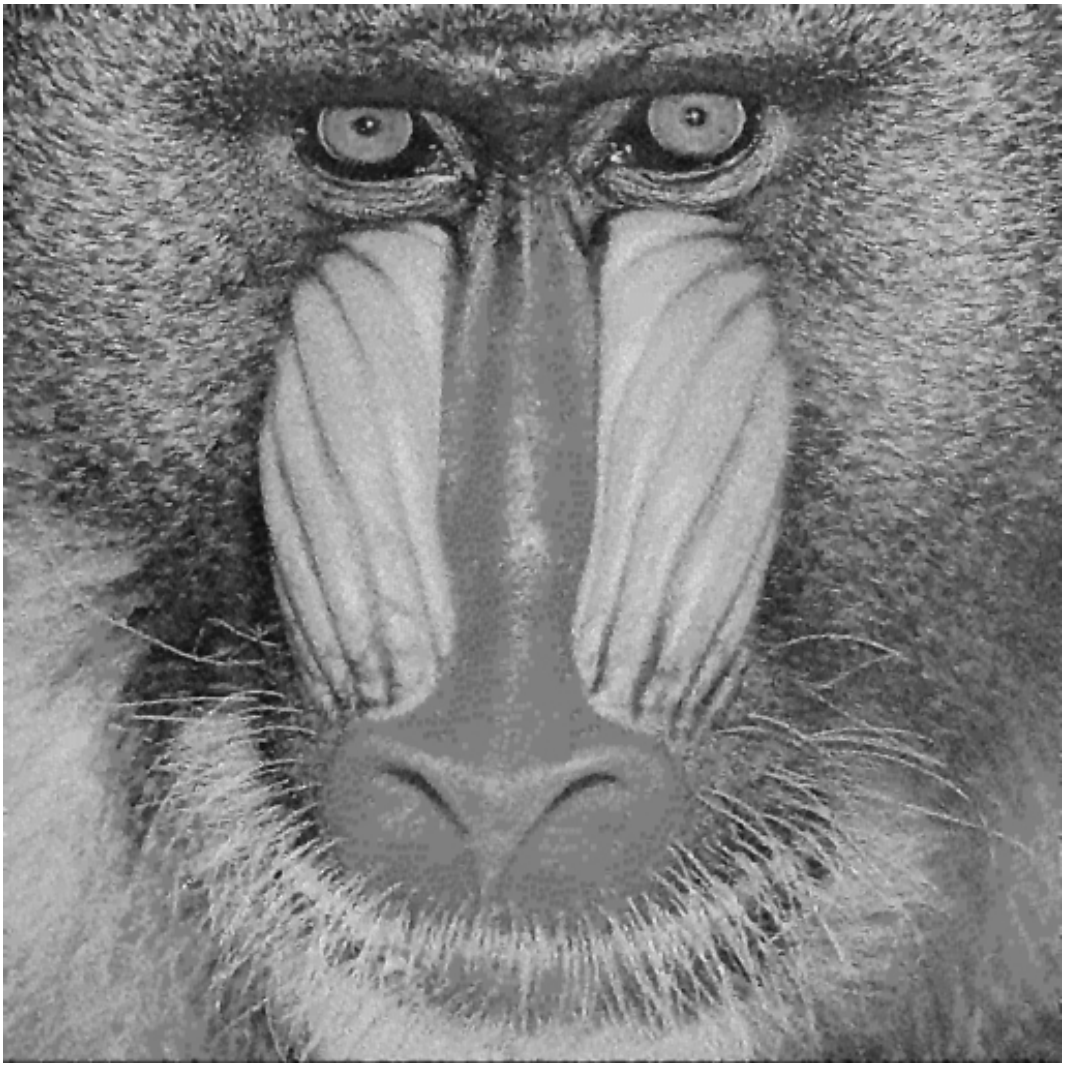

Figure 2.3: Image obtained by using the LUT method.

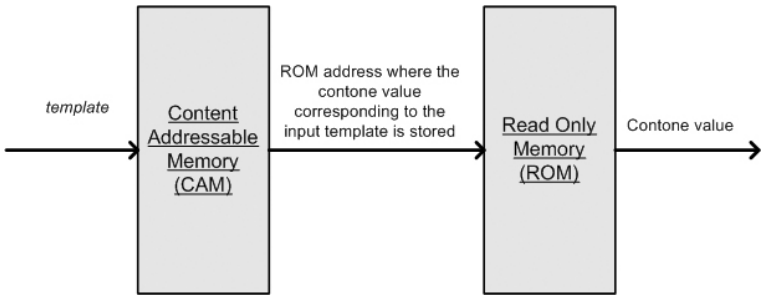

Figure 2.4: Implementation of a Look-Up Table (LUT).

way that templates are fetched from the halftone image and go to the CAM that returns the ROM addresses. The ROM address is now input to the ROM and returns the contone value of the inputted template. When all steps are pipelined then they can occur in parallel on different templates which can be fetched on consecutive clock cycles from the halftone image. Figure 2.4 shows the VLSI implementation of the LUT using CAM-ROM pair.

2.2.2 Hybrid LMS-MMSE Inverse Halftoning

Hybrid LMS (least mean square)-MMSE (minimum mean square error) is another Look-Up Table (*LUT*) method proposed by Pao-Chi Cheng, Che-Sheng Yu, and Tien-Hsu Lee [8]. It uses a hybrid of adaptive filtering and Look-Up Table (LUT). The LUT contains entries for pixel patterns that are obtained through LUT generation using training set images. During inverse halftoning, if a new pattern that was not present in the training set images occurs, then adaptive filtering is used instead of LUT. This method cannot be completely dependent on LUT even if the size of training set is increased. Therefore it employed hybrid of adaptive filtering and LUT. The image quality of this method is illustrated through two images *lena* and *peppers* in

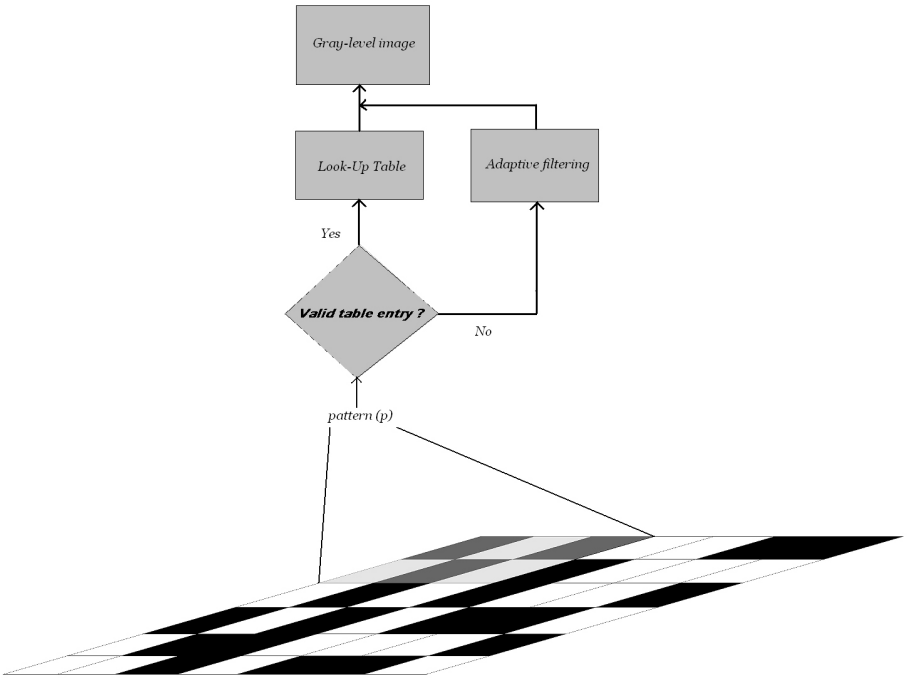

Figure 2.5: Look-Up Table (LUT) inverse halftoning using LMS-MMSE LUT method.

which *lena* has image quality 30.79 dB and *peppers* has image quality 30.67 dB. The procedure of this method is illustrated in Figure 2.5.

2.2.3 Edge-Based Look-Up Table ($ELUT$) Method

Kuo Liang Chung and Shih Tung Wu [9] presented a method to improve the image quality of the previous two LUT methods. It consists of an edge-based Look-Up Table ($ELUT$), that is built from the LUT in any of the two previous methods. The LUT has size that is 39 times that of the LUT of the previous two methods. This method shows that it can improve the performance of the LUT method of Mese and Vaidyanathan by 0.45 dB. The comparison results are shown in Table 1.2. The procedure of this method is illustrated in Figure 2.6.

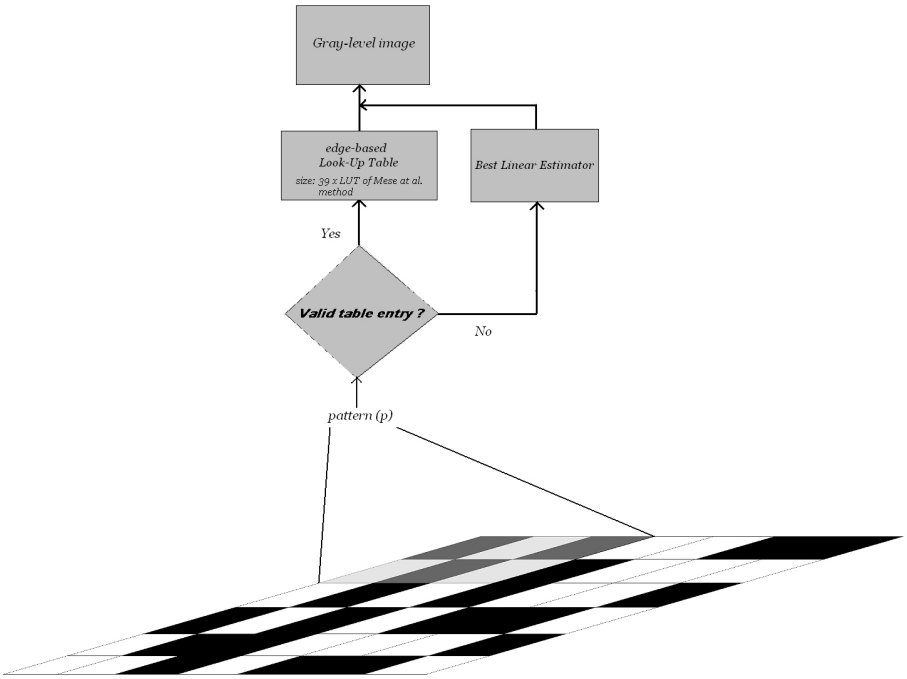

Figure 2.6: Look-Up Table (LUT) inverse halftoning using edge-based LUT ($ELUT$).

2.3 Discussion

We choose LUT method of Mese and Vaidyanathan [6] to be enhanced for parallel LUT inverse halftoning. It is fastest among all other algorithms and offers good image quality. It also has smallest LUT size and can be made computation-free when same halftone algorithm is used in training set images and in halftone images that are going to be inverse halftoned. The other methods either cannot be completely computation-free or have very large LUT.

2.4 Summary

In this chapter we presented several computation and Look-Up Table (LUT) based algorithms to perform inverse halftone operation. Three computation based algo-

rithms are shown that exploit functions from signal processing to accomplish fast, low memory, and less computational inverse halftoning. The LUT inverse halftoning algorithms are very fast and have very less computation. They also have comparable image quality. The LUT method proposed by Mese et al. is the fastest and is completely computation free when same algorithm is used in training set images and in halftone images.

CHAPTER 3

PROPOSED ALGORITHMS

LUT (Look-Up Table) inverse halftoning comprise a single LUT, through which only one pixel can obtain its contone value (or gray level) value simultaneously. Therefore, it is also called serial LUT method. However, if the single LUT is partition into two or more parts through some method then parallel LUT inverse halftoning can be performed because, now two or more pixels can obtained thier contone values simultaneously from different partitions of the LUT. The algorithms proposed to perform parallel LUT inverse halftoning enhance the serial LUT method proposed by Mese et al. [6] and uses N number of Look-Up Tables called smaller Look-Up Tables ($s - LUTs$) in place of a single LUT. They consist of two steps, in the first step the $s - LUTs$ are generated, and in the second step inverse halftone operation is performed. The proposed algorithms have the following unique features:

1. The total entries in the LUT of serial method and entries in all $s - LUTs$ of the proposed algorithms always remains same irrespective to the value of N

2. Proposed algorithms defined several functions that form the basis to develop

algorithms to generate $s-LUTs$ and algorithms to perform parallel LUT inverse halftone operation

3. The parallel methods have significant advantage in speed of inverse halftone operation over the serial LUT method

While all proposed algorithms follow a similar procedure they differ from each other in terms of their computation, image quality and gain in clock cycles they can offer over the serial LUT method. The rest of this chapter describe the proposed algorithms and analysis of their computation.

3.1 Problem Formulation

The serial Look-Up Table (LUT) method is shown in pseudo-code of Figure 3.1. The input to the algorithm is a halftone image represented by $Halftone$ and has lxm dimensions. The output from the algorithm is a gray level image that is obtained after inverse halftoning of the input image and it is represented by $Gray\_level$. The algorithm has two time stamps t_0 and t_1. At t_0 it calls the function *FETCH_TEMPLATE(i,j)* that fetches a template from the halftone image. At second time stamp t_1 it calls the function *LUT(t)* that returns its contone value stored in the LUT. The function *FETCH_TEMPLATE(i,j)* fetches templates that has pixel 0 at location given by i and j in the halftone image. The two *for* loops shown continue until all pixels in the halftone image are fetched and inverse halftoned through the LUT. The template has $p-bits$ where p varies from 16 to 21 based on the type of template used. Please refer to Chapter 2 for description on template types. The two time stamp operations can

Algorithm Serial LUT Method (Halftone(l,m))
(*Halftone(l,m) is the halftone image to be inverse halftoned and have l x m dimensions*)
Begin
$for\ (int\ i = 0; i < l - 1; i++)$
    $for\ (int\ j = 0; j < m - 1; j++)\{$
        $t(0 \cdots p-1) = FETCH\_TEMPLATE(Halftone(i,j))\ at\ time\ t_0$
        Gray_level(i,j)=LUT(t) at time t_1
}
End

Figure 3.1: Serial LUT method of inverse halftoning.

also be pipelined to operate parallel on different data inputs.

The parallel LUT inverse halftoning procedure is described in Figure 3.2. It fetches up to k templates ($t_0,\ t_1$ to t_k) from the halftone image at time stamp t_0. It is performed by parallel calling functions *FETCH_TEMPLATE(i,j)*, *FETCH_TEMPLATE(i+1,j)* to *FETCH_TEMPLATE(i+k-1,j)*. At time stamp t_1 it calls function $FIND\_s - LUT\_NUMBER(t_k)$ that returns the $s-LUT$ where the template t_k goes to fetch its contone value. This function is also called in parallel to all templates. At time stamp t_2 continuous tone values of fetched templates are retrieved from the corresponding $s-LUT$ by parallel calling functions $s-LUT_0(t_0)$, $s-LUT_1(t_1)$ to $s-LUT_{N-1}(t_{n-1})$. In this way parallel LUT method can speed up the inverse halftone operation by up to n times over the serial LUT method. It is also pipelined in which operations at each time stamp can occur concurrently on different data. The actual algorithms to perform parallel LUT inverse halftoning require more sophistication. These are discussed in the following sections.

Algorithm Parallel LUT Method (Halftone(l,m), N, k)
(*Halftone(l,m) is the halftone image to be inverse halftoned and have $l \cdot m$ dimensions*)
(*N is the number of smaller Look-Up Tables ($s - LUT$)*)
(*k is the number of concurrently fetched templates templates*)
Begin
$for\ (int\ i = 0; i < l - 1; i++)$
    $for\ (int\ j = 0; j < m - 1; j++)\{$
    $for\ (int\ n = 0; n < k; n++)\{$
        $t_n(0 \cdots p - 1) = FETCH\_TEMPLATE(Halftone(i, j + k))\ at\ time\ t_0\ \}$
    $for\ (int\ n = 0; n < k; n++)\{$
        $r_k = FIND\_s - LUT\_NUMBER(t_k)$ at time t_1
    $for\ (int\ n = 0; n < k; n++)\{$
        Gray_level(i,j)=$s - LUT_{r_n}(t)$ at time t_2
}
End

Figure 3.2: Parallel Look-Up Table (LUT) inverse halftoning.

3.2 Proposed Functions

In parallel Look-Up Table (LUT) inverse halftoning k templates are fetched from the halftone image simultaneously. They must go to distinct $s - LUTs$ to obtained their contone values. The first task in this regard is to uniquely represent all concurrently fetched templates from each other. It is accomplished by defining several functions that return unique numbers in range from 0 to $N - 1$ for each template, where N is the total number of $s - LUTs$. The following four functions are defined:

1. Relative XOR Change (RXC) function,

2. XOR with Mean (XM) function,

3. Only Addition (OA) function, and

4. Mod Only (MO).

The following text describe each function in detail.

3.2.1 Function Relative XOR Change (RXC)

This function is motivated when some halftone images are observed and it is found that adjacent, either top-bottom, or left-right templates are distinct from each other in terms of number of ones and positions of ones and zeros. Based on this observation the function RXC is defined that consists of XOR operation that works on difference between the two templates and then a result is obtained that varies from 0 to $N-1$. The RXC function is shown in Figure 3.3. It has inputs for two templates t_0 and t_1, where t_0 is the template on which the function is going to be applied and t_1 is its neighboring template that is also fetched from the halftone image. The templates t_0 and t_1 can belong to any but same template types. The first step in the function is to apply XOR operation between t_0 and t_1. In the next step, bits in the XOR result are added to find the number of ones in it. The final step is to keep only $log_2 N$ least significant bits of the result from the previous step. It is also the result of RXC function. In this function N should be an exponent of 2 i.e., $N = 2^i$ where $i \in$ whole Numbers.

3.2.2 Function XOR with Mean (XM)

This function is proposed to eliminate the ambiguity in RXC function, that is, RXC yields a different result for the template that occurs its next time in the training set.

Function RXC($t_0(0 \cdots p-1), t_1(0 \cdots p-1), N$);
(*$t_0(0 \cdots p-1)$ is the template to be inverse halftoned and have p-bits*)
(*$t_1(0 \cdots p-1)$ is the neighboring template of t_0)
(*N is the number of smaller Look-Up Tables (s-LUTs) and $N \in$ Whole Numbers*)
Begin
$r_0(0 \cdots p-1) = t_0(0 \cdots p-1)\ XOR\ t_1(0 \cdots p-1)$
$s_0(0 \cdots log_2 p-1) = r_0(0) + r_0(1) + r_0(2) + \cdots + r_0(p-1)$
$y(0 \cdots log_2 N-1) = s_0(0 \cdots log_2 N-1)$
return $y(0 \cdots log_2 N-1)$
End

Figure 3.3: Illustration of function Relative XOR Change (RXC).

In this function XOR operation is performed with m and consequently same result is returned for a template value. The value of m is calculated by taking mean of all templates present in the training set. The description of training set is shown with description of algorithms proposed to generate smaller Look-Up Tables ($s-LUTs$). This function is shown in Figure 3.4. The first step is to take XOR between t_0 and m. In the next step the bits in the XOR result are added to find the number of ones in it. Finally, only $log_2 N$ least significant bits are kept that is the result of this function.

3.2.3 Function Only Addition (OA)

This function is similar to XM function except that it has no m input and has no step to perform XOR operation. It only accepts one template t_0. The first step is to add the bits in t_0 to find the number of ones it it. Finally only $log_2 N$ bits are kept to yield the result. This function is shown in Figure 3.5.

Function XM($t_0(0 \cdots p-1), N, m$);
(*$t_0(0 \cdots p-1)$ is the input template having p-bits*)
(*N is the number of smaller Look-Up Tables (s-LUTs) and $N \in$ Whole Numbers*)
(*m is the mean of entries in all smaller Look-Up Tables (s-LUTs)*)
Begin
$r_0(0 \cdots p-1) = t_0(0 \cdots p-1)\ XOR\ m(0 \cdots p-1)$
$s_0(0 \cdots log_2 p-1) = r_0(0) + r_0(1) + r_0(2) + \cdots + r_0(p-1)$
$y(0 \cdots log_2 N-1) = s_0(0 \cdots log_2 N-1)$
return $y(0 \cdots log_2 N-1)$
End

Figure 3.4: Illustration of function XOR with Mean (XM).

Function OA($t_0(0 \cdots p-1), N$);
(*$t_0(0 \cdots p-1)$ is the input template having p-bits*)
(*N is the number of smaller Look-Up Tables (s-LUTs)*)
Begin
$s_0(0 \cdots log_2 p-1) = t_0(0) + t_0(1) + t_0(2) + \cdots + t_0(p-1)$
$y(0 \cdots log_2 N-1) = s_0(0 \cdots log_2 N-1)$
return $y(0 \cdots log_2 N-1)$
End

Figure 3.5: Illustration of function Only Addition (OA).

Function MO($t_0(0 \cdots p-1), N$);
(*$t_0(0 \cdots p-1)$ is the input template having p-bits*)
(*N is the number of smaller Look-Up Tables (s-LUTs)*)
Begin
$y(0 \cdots log_2N - 1) = t_0(0 \cdots log_2N - 1)$
return $y(0 \cdots log_2N - 1)$
End

Figure 3.6: Illustration of function Mod Only (MO).

3.2.4 Function Mod Only (MO)

In this function no computation is performed at all. Only least significant $log_2N - bits$ of fetched templates are used to give the number of $s - LUTs$. This function is motivated by the same observation as shown in *Function RXC* that adjacent templates differ from each other and their least $log_2N - bits$ also differ from each other. The function is described in Figure 3.6.

3.3 Performance of Proposed Functions

To measure the extent to which one function is able to uniquely represent two or more concurrently fetched templates from each other the proposed functions are implemented in Java programming language. The program inputs halftone images and yield the percentage of times two or more templates have same $s - LUT$ numbers among their concurrently fetched templates. The results are obtained on images: *peppers*, *trees*, and *clock* and an average of all images is taken at the end to show one value. The results are illustrated using graphs in which $x - axis$ shows the number

of $s - LUT$ i.e., N, and $y - axis$ shows the *Percentage of times the given function yields same result for more than one function.* Result of each function is shown in a separate graph. In Figure 3.7, performance of function RXC is shown. Three plots are drawn for k= 4, k= 8 and k= 16. It is observed that best results are obtained when k is small. It is also observed that increase in value of N improves results but after N= 16 there is no further improvement in the results. The performance of XM function is shown in Figure 3.8. The values along $x - axis$ and $y - axis$ represent same quantities as in the graph of Figure 3.7. The graph again shows that best results are obtained when k is small i.e., $k = 4$ and there is no improvement is results after $k = 16$. The performance of OA function is shown in Figure 3.9 and they also follow the same behavior as shown by previous two graphs. The graph in Figure 3.10 shows the performance of MO function and x and y axises have the same terms as other graphs. It shows an improvement in results, i.e., less percentage of templates have duplicate results when the value of k is small i.e., $k = 4$ and only very little improvement occurs when value of N is increased from 16 to 32.

3.4 Algorithms for smaller Look-Up Table ($s - LUT$) Generation

The algorithms to generate smaller Look-Up Tables ($s - LUT$) are developed using functions Xor with Mean (XM), Only Addition (OA), Mod Only (MO) and Relative Xor Change (RXC). The general procedure adopted in $s - LUT$ generation is that

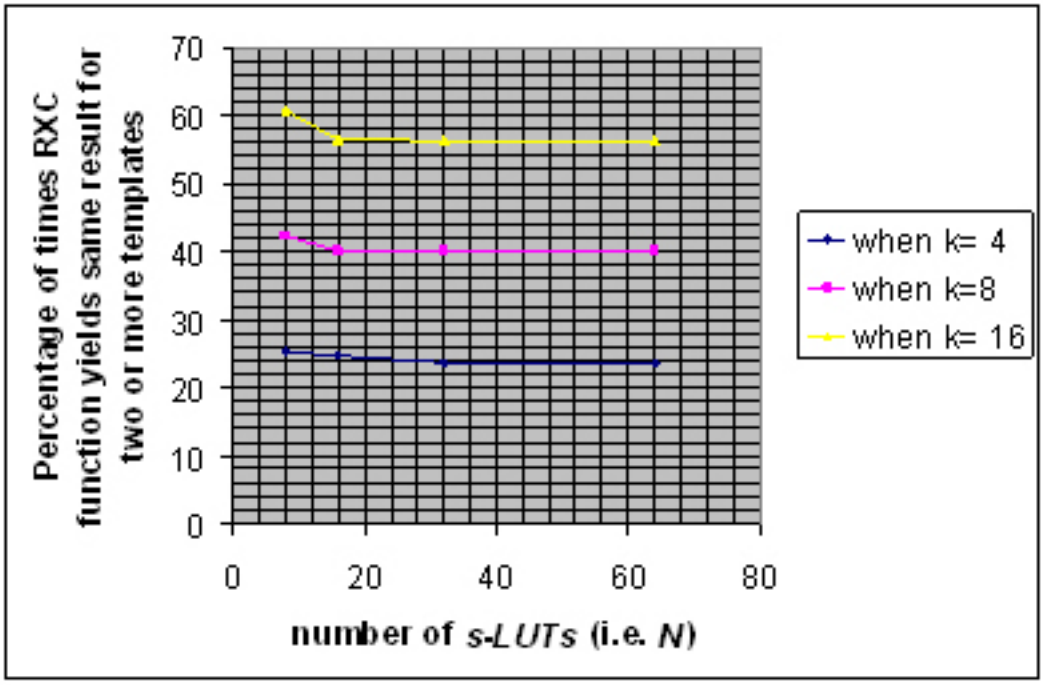

Figure 3.7: Graph showing performance of RXC function.

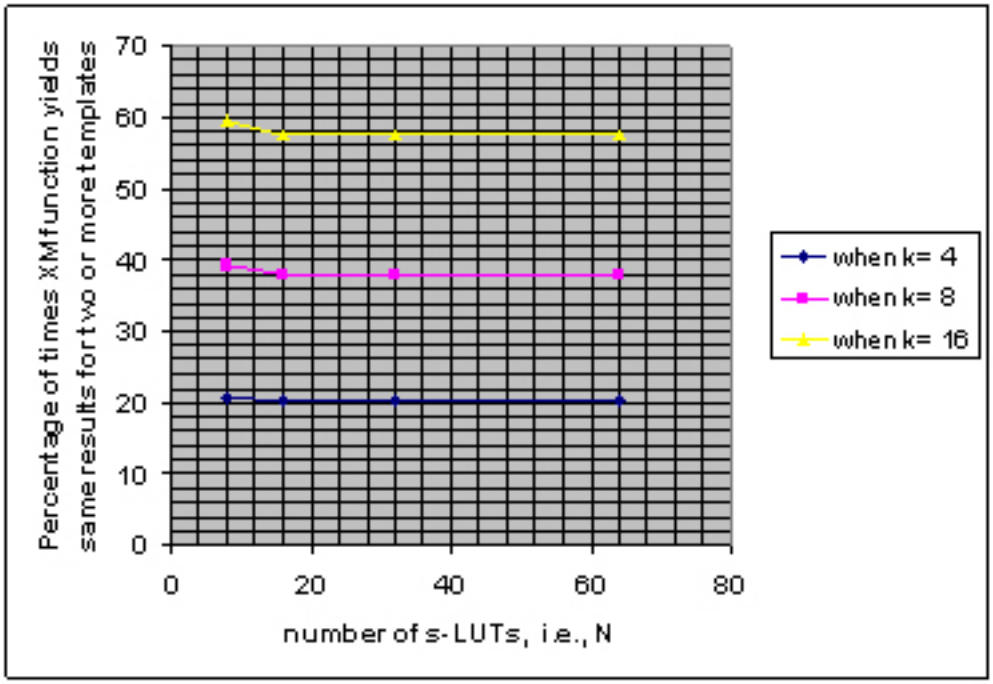

Figure 3.8: Graph showing performance of XM function.

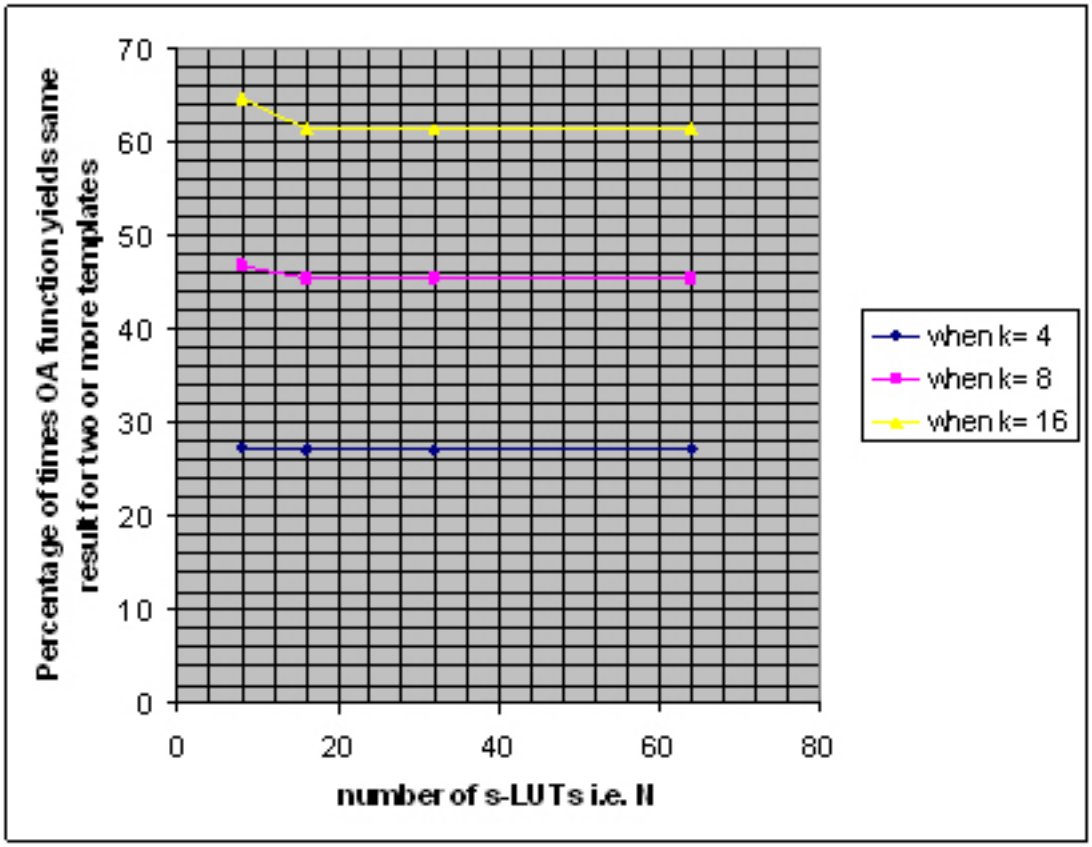

Figure 3.9: Graph showing performance of OA function.

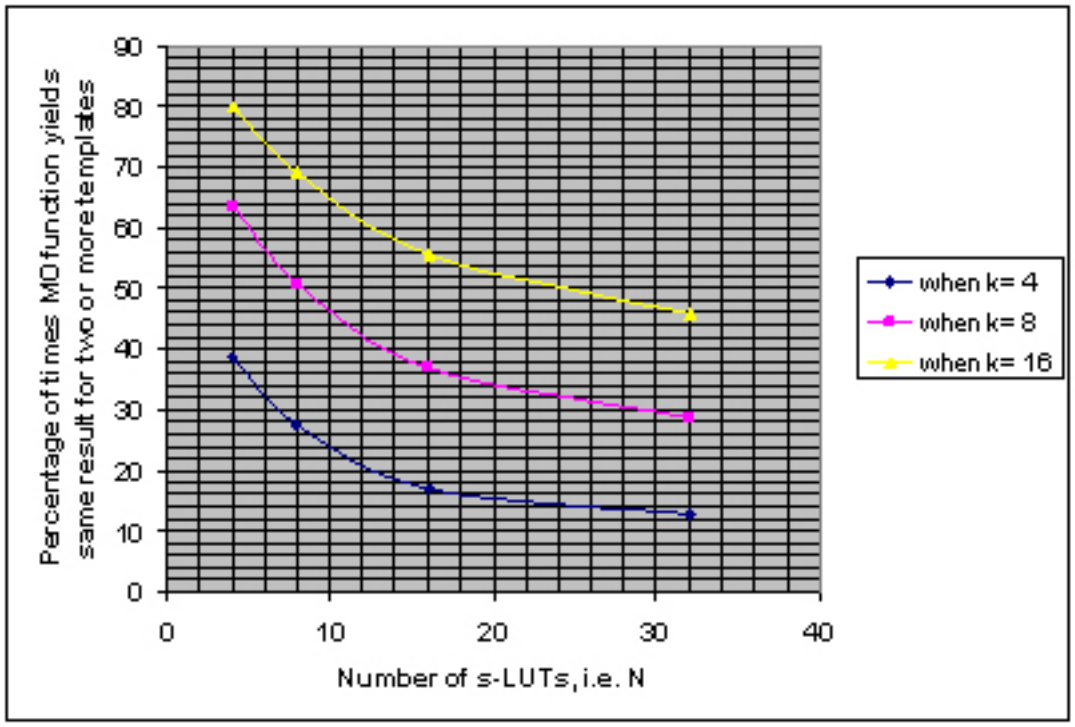

Figure 3.10: Graph showing performance of MO function.

first value of N is chosen. Then a training set is build that consists of continuous tone images and corresponding halftone images. The halftone images are obtained from continuous tone images by running a halftone algorithm like Floyd and Steinberg [10] on them. The gray level images in the training set are called $Graytrain$ and halftone images in the set are called $Halftrain$. The steps of the algorithm are shown below:

1 The value of N is chosen and smaller Look-Up Tables ($s - LUT$) are numbered from 0 to $N-1$. N should be a multiple of 2 i.e., $N = 2^i$ where $i = \in$ Whole Numbers.

2 Fetch one template from the set $Halftrain$ and also fetch its contone value from set $Graytrain$.

3 Check if the fetched template already exists in any smaller Look-Up Table

$(s - LUT)$.

4 If yes then add the continuous tone value to the already existing value and store the value of number of times the same template has occurred.

5 If the template is not in any $s - LUT$ then apply function XM, OA, MO, or RXC to the fetched template.

6 Now add the template and its contone value to the smaller Look-Up Table $(s - LUT)$ that has same number as the value returned by the XM, OA, MO or RXC function.

7 Repeat the above procedure until all templates are fetched from all images in the set.

8 For all templates that are stored in their smaller Look-Up Tables $(s - LUT)$ divide their contone values with the number of times the same template has occurred.

In the above algorithm either XM, OA, MO, or RXC function should be used at a time. The algorithms for parallel LUT inverse halftoning should use the same function as used in $s - LUT$ generation and same value of N as well. For example, when the $s - LUT$ are generated using XM function and with $N = 8$, then parallel LUT inverse halftoning must be performed using XM_PL or XM_NPL algorithms with $N = 8$.

3.5 Algorithms for Parallel LUT Inverse Halftoning

Five algorithms are proposed that can perform parallel LUT inverse halftone operation. They differ from each other in terms of image quality and clock cycle gain over the serial LUT method. The four algorithms are:

Algorithm-1 XM (XOR with Mean) with Pixel Loss (XM_PL)

Algorithm-2 XM (XOR with Mean) with No Pixel Loss (XM_NPL)

Algorithm-3 OA (Only Addition) with Pixel Loss (OA_PL)

Algorithm-4 OA (Only Addition) with No Pixel Loss (OA_NPL)

Algorithm-5 MO (Mod Only) with Pixel Loss (MO_PL)

Algorithm-6 MO (Mod Only) with No Pixel Loss (MO_NPL)

Algorithm-7 RXC (Relative XOR Change) with Pixel Loss (RXC_NPL)

3.5.1 Xor with Mean with Pixel Loss (XM_PL) Algorithm

This algorithm uses XM function to perform parallel LUT inverse halftoning. However, it also requires $s - LUTs$ generated using XM function with same value of N. It is shown in Figure 3.11. The first step is to fetch up-to k templates from the halftone image, then XM function is applied to all templates. This is followed by matching if any two, or more, templates return same values from XM functions. If so then

only one template having highest index i, where i varies from 0 to $k-1$ is kept while other templates having same result from XM function are discarded. For example when t_3 and t_2 return same result from XM function then t_2 is discarded. After this non-discarded templates go to $s-LUT$ and obtain their contone values. Now the templates that were discarded are assigned contone values of their nearest non-zero higher index template. That is in t_2, t_3 example t_2 is assigned contone value held by t_3 and no change in contone value of t_3 occurs. All iterations inside the for loop are executed concurrently. Each for loop can run concurrently on different data inputs in this way the algorithm is pipelined. Therefore, new k templates can be fetched from the halftone image on every clock cycle and go through inverse halftone operation. The output of the algorithm is represented by $Gray\_level$ that is a $2D$ array representing continuous tone image obtained after inverse halftone operation.

3.5.2 Clock Cycles Consumed in XM_PL Algorithm

The clock cycles consumed in performing parallel LUT inverse halftoning using XM with Pixel Loss (XM_PL) algorithm are shown below:

1. $Number\ of\ pixels\ in\ the\ halftone\ image = l \cdot m$
2. $Number\ of\ concurrently\ fetched\ templates = n$
3. $CC\ (Clock\ Cycles)\ consumed\ in\ serial\ LUT\ Method = l \cdot m$
4. $CC\ in\ parallel\ LUT\ Method = 2 \cdot (pipeline\ stages) + \dfrac{l \cdot m}{n}$
5. $Percentage\ gain\ in\ CC\ over\ serial\ LUT\ method = \dfrac{2 \cdot (pipeline\ stages)}{l \cdot m} + \dfrac{1}{n}$

Algorithm XM_PL($Halftone\_image, N, k$);
(*$Halfone\_image$ is the input halftone image*)
(*N is the number of smaller Look-Up Tables (s-LUTs)*)
(*k is the number of concurrent templates*)
Begin
for (i=0;i< image_height;i++)
for (j=0;j< image_width;j++) {
for (n=0; $< k$**; n++)**{
t_n=Call Fetch_Template(i,j+k) }
for (n=0; n< k**; n++)**{
r_n=Call XM(i,j+n) }
for (m=k;m>1;m–) {
sel=r_m
for (p=m;p>0;p–) {
if sel==r_p
Call DISCARD(r_p) } }
for (n=0; n< k**; n++)** {
Gray_Image(i,j+n)=Call $s - LUT_{r_n}(t_n)$ }
for (n=0; n< $k - 1$**; n++)** {
if r_n was DISCARDED
r_n=r_{n+1} }
End

Figure 3.11: Illustration of XM with Pixel Loss (XM_PL) algorithm.

3.5.3 Xor with Mean with No Pixel Loss (XM_NPL) Algorithm

This algorithm also employs function XM to perform parallel LUT inverse halftone operation. However, it does not discard any templates like XM_PL algorithm. It is shown in Figure 3.12. The first step is to fetch k templates from the halftone image. In the next step, k instances of function XM are applied concurrently on fetched templates. In this algorithm a queue of depth $= k$ templates is present before each $s - LUT$. The queues accepts parallel input of size k templates and yields one non-zero template at a time until all parallel in template are yielded to output. The output starts from next clock cycle in which input is received. The queues have a signal $FINISH$ that is high until queues are non-empty. The algorithm cannot fetch new templates from the halftone image and registers cannot obtain new values until $FINISH$ of any queue is high. The queues inputs are connected to templates coming after XM function application and output goes to corresponding $s - LUT$. This algorithm is also pipeline i.e., each step can occur in parallel on different inputs. However, when $FINISH$ signal is high pipeline stalls are introduced that counts to overall clock cycles consumed in inverse halftone operation. The output from this algorithm is also a gray level image represented by $Gray\_level$.

Algorithm XM_NPL($Halftone\_image, N, n, m$);
(*$Halfone\_image$ is the input halftone image*)
(*N is the number of smaller Look-Up Tables (s-LUTs)*)
(*k is the number of concurrent templates*)
(*m is the number of mean of entries in all s-LUTs*)
Begin
for (i=0;i< image_height;i++)
for (j=0;j< image_width;j++) {
for (k=0; k< n**; k++)**{
t_k=Call Fetch_Template(i,j+k) }
for (n=0; n< k**; n++)**{
r_n=Call XM(i,j+n) }
for (n=0; n< k**; n++)**{
$Queue_{r_n}.Insert(t_n)$ }
for (n=0; n< k**; n++)**{
Gray_level(i,j+n)=$s - LUT_{r_n}(Queue_{r_n}.Out())$ }
}
End

Figure 3.12: Illustration of XM with No Pixel Loss (XM_NPL) algorithm.

3.5.4 Clock Cycles Consumed in XM_NPL Algorithm

The clock cycles consumed in performing parallel LUT inverse halftoning using XM_NPL are computed below:

1. $Number\ of\ pixels\ in\ the\ halftone\ image = l \cdot m$
2. $Number\ of\ concurrently\ fetched\ templates = n$
3. $CC\ (Clock\ Cycles)\ consumed\ in\ serial\ LUT\ Method = l \cdot m$
4. $Average\ pipeline\ stalls\ per\ fetching = q$
5. $CC\ in\ parallel\ LUT\ Method = 2 \cdot (pipeline\ stages) + \frac{l \cdot m}{n}(1 + q)$
6. $Percentage\ Gain\ in\ CC\ over\ serial\ LUT\ method = \frac{2 \cdot (pipeline\ stages)}{l \cdot m} + \frac{1}{n}(1+q)$

3.5.5 Only Addition with Pixel Loss (OA_PL) Algorithm

This algorithm is same as XM_PL algorithm except that function OA is applied in place of XM. The algorithm is shown in Figure 3.13. The clock cycles consumed in inverse halftone operation are also equal to clock cycles consumed in XM_PL algorithm. Please refer to the description of XM_PL algorithm for detailed explanation of the algorithm.

3.5.6 Only Addition with No Pixel Loss (OA_NPL) Algorithm

This algorithm is also same as XM_NPL except that function OA is applied in place of function XM function. The clock cycles consumed to perform parallel LUT inverse

Algorithm OA_PL($Halftone\_image, N, n$);
($*Halfone\_image$ is the input halftone image*)
($*N$ is the number of smaller Look-Up Tables (s-LUTs)*)
($*k$ is the number of concurrent templates*)
Begin
for (i=0;i< image_height;i++)
for (j=0;j< image_width;j++) {
for (n=0; n< k**; n++)**{
t_n=Call Fetch_Template(i,j+n) }
for (n=0; n< k**; n++)**{
r_n=Call OA(i,j+n) }
for (m=k;m>1;m–) {
sel=r_m
for (p=m;p>0;p–) {
if sel==r_p
Call DISCARD(r_p) } }
for (n=0; k< k**; n++)** {
Gray_Image(i,j+n)=Call $s - LUT_{r_n}(t_n)$ }
for (n=0; n< $k-1$**; n++)** {
if r_n was DISCARDED
r_n=r_{n+1} }
End

Figure 3.13: Illustration of OA with Pixel Loss (OA_PL) algorithm.

Algorithm OA_NPL($Halftone\_image, N, n, m$);
(*$Halfone\_image$ is the input halftone image*)
(*N is the number of smaller Look-Up Tables (s-LUTs)*)
(*k is the number of concurrent templates*)
(*m is the number of mean of entries in all s-LUTs*)
Begin
for (i=0;i< image_height;i++)
for (j=0;j< image_width;j++) {
for (n=0; n< k**; n++)**{
t_n=Call Fetch_Template(i,j+n) }
for (n=0; n< k**; n++)**{
r_n=Call OA(i,j+n) }
for (n=0; n< k**; n++)**{
$Queue_{r_n}.Insert(t_n)$ }
for (n=0; n< n**; n++)**{
Gray_level(i,j+n)=$s - LUT_{r_n}(Queue_{r_n}.Out())$ }
}
End

Figure 3.14: Illustration of OA with No Pixel Loss (OA_NPL) algorithm.

halftoning using this method are same as consumed in XM_NPL algorithm. It is shown in Figure 3.14. Explanation of this algorithm can be found in description of XM_NPL algorithm.

3.5.7 Mod Only with Pixel Loss (MO_PL) Algorithm

This algorithm is similar to XM_PL and OA_PL algorithms except that function MO is used in it. The procedure is listed in Figure 3.15.

Algorithm $MO\_PL(Halftone\_image, N, n)$;
(*$Halfone\_image$ is the input halftone image*)
(*N is the number of smaller Look-Up Tables (s-LUTs)*)
(*k is the number of concurrent templates*)
Begin
for (i=0;i< image_height;i++)
for (j=0;j< image_width;j++) {
for (n=0; n< k**; n++)**{
t_n=Call Fetch_Template(i,j+n) }
for (n=0; n< k**; n++)**{
r_n=Call MO(i,j+n) }
for (m=k;m>1;m–) {
sel=r_m
for (p=m;p>0;p–) {
if sel==r_p
Call DISCARD(r_p) } }
for (n=0; k< k**; n++)** {
Gray_Image(i,j+n)=Call $s - LUT_{r_n}(t_n)$ }
for (n=0; n< $k-1$**; n++)** {
if r_n was DISCARDED
r_n=r_{n+1} }
End

Figure 3.15: Illustration of MO with Pixel Loss (MO_PL) algorithm.

Algorithm MO_NPL($Halftone\_image, N, n, m$);
(*$Halfone\_image$ is the input halftone image*)
(*N is the number of smaller Look-Up Tables (s-LUTs)*)
(*k is the number of concurrent templates*)
(*m is the number of mean of entries in all s-LUTs*)
Begin
for (i=0;i< image_height;i++)
for (j=0;j< image_width;j++) {
for (n=0; n< k**; n++)**{
t_n=Call Fetch_Template(i,j+n) }
for (n=0; n< k**; n++)**{
r_n=Call MO(i,j+n) }
for (n=0; n< k**; n++)**{
$Queue_{r_n}.Insert(t_n)$ }
for (n=0; n< n**; n++)**{
Gray_level(i,j+n)=$s - LUT_{r_n}(Queue_{r_n}.Out())$ }
}
End

Figure 3.16: MO with No Pixel Loss (MO_NPL) algorithm.

3.5.8 Mod Only with No Pixel Loss (MO_NPL) Algorithm

This algorithm is similar to XM_NPL and OA_NPL algorithms except that function MO is called in it. The procedure of this function is shown in Figure 3.16.

3.5.9 Relative Xor Change with Pixel Loss (RXC_PL) Algorithm

This algorithm uses RXC function to perform parallel LUT inverse halftoning. It has procedure similar to XM_PL algorithm and is shown in Figure **??**. In which all steps are similar to XM_PL algorithm except that RXC function is applied instead

Algorithm RXC_PL($Halftone\_image, N, n$);
(*$Halfone\_image$ is the input halftone image*)
(*N is the number of smaller Look-Up Tables (s-LUTs)*)
(*k is the number of concurrent templates*)
Begin
for (i=0;i< image_height;i++)
for (j=0;j< image_width;j++) {
for (n=0; n< k**; n++)**{
t_n=Call Fetch_Template(i,j+n) }
for (n=0; n< k**; n++)**{
r_n=Call RXC(i,j+n) }
for (m=k;m>1;m–) {
sel=r_m
for (p=m;p>0;p–) {
if sel==r_p
Call DISCARD(r_p) } }
for (n=0; n< k**; n++)** {
Gray_Image(i,j+n)=Call $s - LUT_{r_n}(t_n)$ }
for (n=0; n< $k-1$**; n++)** {
if r_n was DISCARDED
r_n=r_{n+1} }
End

Figure 3.17: Illustration of RXC with Pixel Loss (RXC_PL) algorithm.

of XM function. Please refer to the description of XM_PL algorithms for details on steps of the algorithm.

3.6 Template Values that are Not Present in the Training Set Images

In the generation of $s-LUTs$ the contents are generated from a training set comprising of continuous tone images and corresponding halftone images. The values of templates

that do not occur in the training set are not stored in any $s-LUT$ and consequently their contone values are also not stored in any $s-LUT$. Such templates are called *non-existence templates*. When *non-existence templates* occur during inverse halftoning process, their contone values are determined by copying contone values from their neighboring templates that are also fetched concurrently from the halftone image. The serial LUT method proposed a computational method for *non-existence templates* however, the proposed algorithms simply copy contone values of neighboring templates to their place.

3.7 Summary

In this chapter we showed proposed algorithms to perform parallel LUT inverse halftoning and they comprise three components: (1) Functions to help develop algorithms for $s-LUT$ generation and parallel LUT inverse halftoning, (2) Algorithms to generate $s-LUT$, and (3) Algorithms to perform parallel LUT inverse halftoning. The functions proposed are: XOR with Mean (XM), Only Addition (OA), and Relative XOR Change (RXC). The algorithms developed to perform parallel LUT inverse halftoning consists of proposed functions as fundamental unit. There are seven algorithms presented and are named as: Xor with Mean with Pixel Loss (XM_PL), Xor with Mean with No Pixel Loss (XM_NPL), Only Addition with Pixel Loss (OA_PL), Only Addition with No Pixel Loss (OA_NPL), Mod Only with Pixel Loss (MO_PL), Mod Only with No Pixel Loss (MO_NPL) and Relative Xor Change with Pixel Loss (RXC_PL) algorithms. XM_PL, OA_PL, MO_PL, and RXC_PL algorithms have

constant gain in clock cycles over the serial LUT method and their image quality varies with values of k and N. Similarly, XM_NPL, OA_NPL, and MO_NPL algorithms have image quality same as serial LUT method but they have varying gain in clock cycles over the serial LUT method.

CHAPTER 4

SIMULATION OF THE PROPOSED ALGORITHMS

In this chapter the proposed algorithms are implemented in software and results are obtained. The software implementation is performed by developing a toolbox in Java programming language that can perform the following tasks:

1. Measure image quality in terms of Peak Signal to Noise Ratio (PSNR) of gray level images obtained from the proposed algorithms. The PSNR is shown in terms of decibels.

2. Measure clock cycles consumed in parallel LUT inverse halftone operation and also measure gain in clock cycles consumed in parallel LUT inverse halftone operation over serial LUT method.

3. Display gray level images obtained from the proposed algorithms.

The selection of Java provides the following advantages:

1. Java enables algorithms based on arrays and link list data structures to form the implementation of proposed algorithms.

2. Creating local objects so that they become unreferenced when that particular task finishes. In Java unreferenced objects are automatically subject to removal from memory using Garbage Collection.

3. Fast program execution allows inputting images of large sizes like 512x512 pixels.

The inverse halftone operation can be performed by first building the Look-Up Table (LUT). The continuous tone images and corresponding halftone images are stored in a directory. The user through the GUI of the toolbox enters the size of the training set i.e., the number of images in the directory. Then the toolbox fetches the templates and corresponding contone values from images in the training set and prepares the LUT. After completing LUT generation serial LUT method or the proposed Xor with Mean with Pixel Loss (XM_PL), Xor with Mean with No Pixel Loss (XM_NPL), Only Addition with Pixel Loss (OA_PL), Only Addition with No Pixel Loss (OA_NPL), Mod Only with Pixel Loss (MO_PL), or Mod Only with No Pixel Loss (MO_NPL) can be executed on test images. All test images are not present in the training set and therefore they could also contain template values that do not occur in the training set images. The toolbox interface allows setting values of k and N before running any particular algorithm. It returns image quality in terms of PSNR for algorithms including serial LUT method, XM_PL, OA_PL, and MO_PL because the gain in clock cycles is constant. It returns gain in clock cycles over the serial LUT methods for algorithms XM_NPL, OA_NPL, and MO_NPL because the image quality is

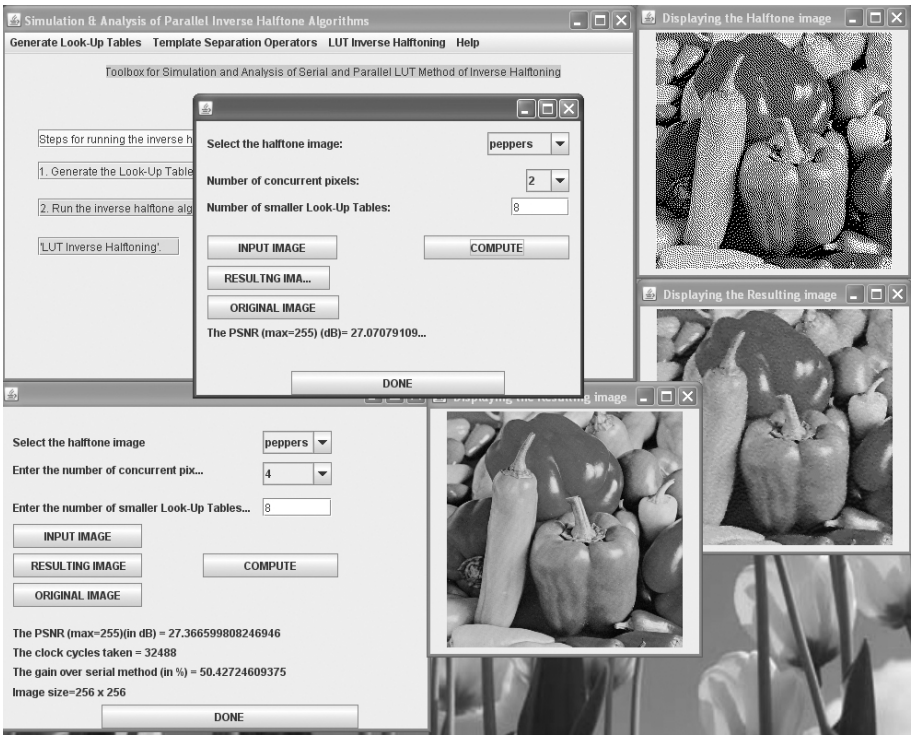

Figure 4.1: Screen shot of the developed simulation and analysis toolbox.

same as serial LUT method. In Figure 4.1, screen shot of a user running the toolbox in Windows XP is shown.

4.1 Simulation of Algorithms to Generate smaller Look-Up Tables ($s-LUTs$)

The training set required to build smaller Look-Up Tables ($s-LUTs$) comprises of continuous tone images and corresponding halftone images. The total number of images in the set is equal to 17. The $s-LUTs$ are generated using functions Xor with Mean (XM), Only Addition (OA), Mod Only (MO), and Relative Xor Change (RXC). The sizes of $s-LUTs$ are also calculated for further computation, that is

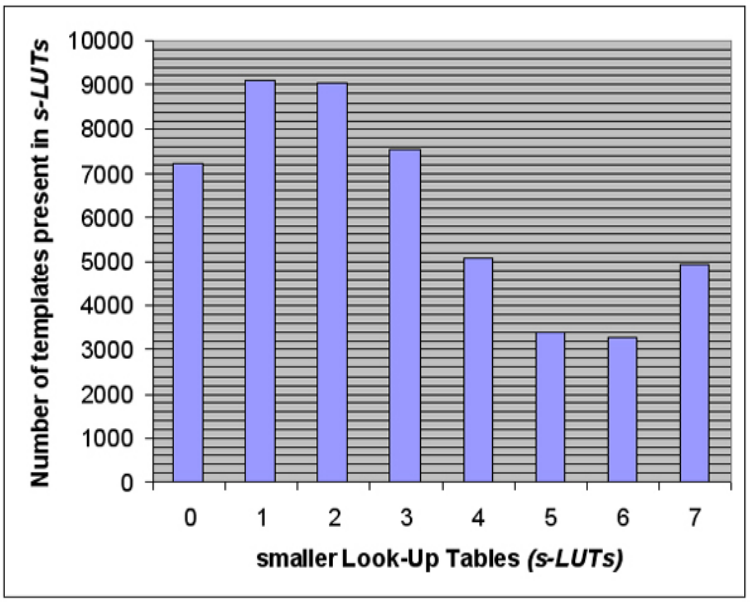

Figure 4.2: Graph showing partitioing of templates to $s-LUTs$ when Xor with Mean (XM) function with $N = 8$ is used.

performed when functions are evaluated in their ability to create partitions of equal sizes. The results are shown using graphs. The graph in Figure 4.2 shows partitioning of templates to $s - LUTs$ when $N = 8$ and XM function is used. The graph in Figure 4.3 shows the table partitioning using OA function with $N = 16$. The graph in Figure 4.4 shows partitioning of templates among eight (i.e., $N = 8$) $s - LUTs$ when OA function is used and Figure 4.5 shows the graph when $N = 16$ and OA function is used again. Finally, distribution using MO function is shown in Figures 4.6 and 4.7. Comparing all graphs show that XM has most uniform partitioning of templates to $s - LUTs$ when $N = 8$ and when $N = 16$ then MO algorithm has most uniform partitioning.

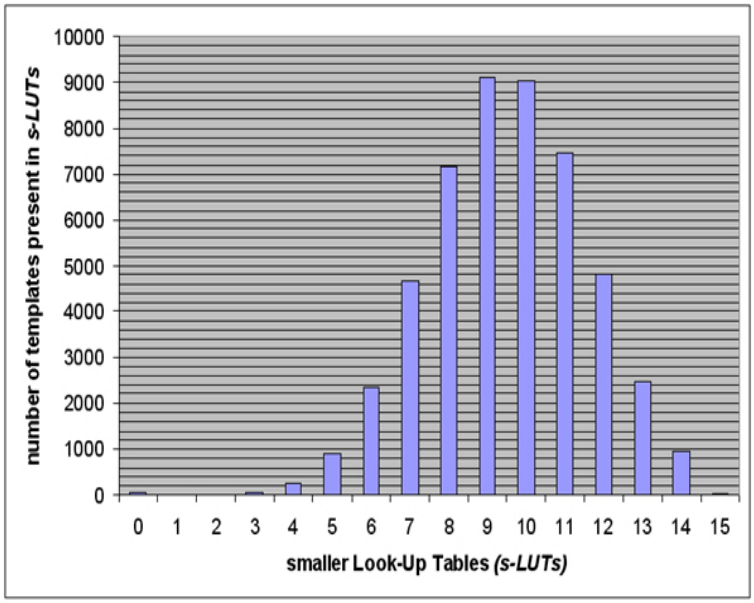

Figure 4.3: Graph showing partitioning of templates to $s - LUTs$ when Xor with Mean (XM) function with $N = 16$ is used.

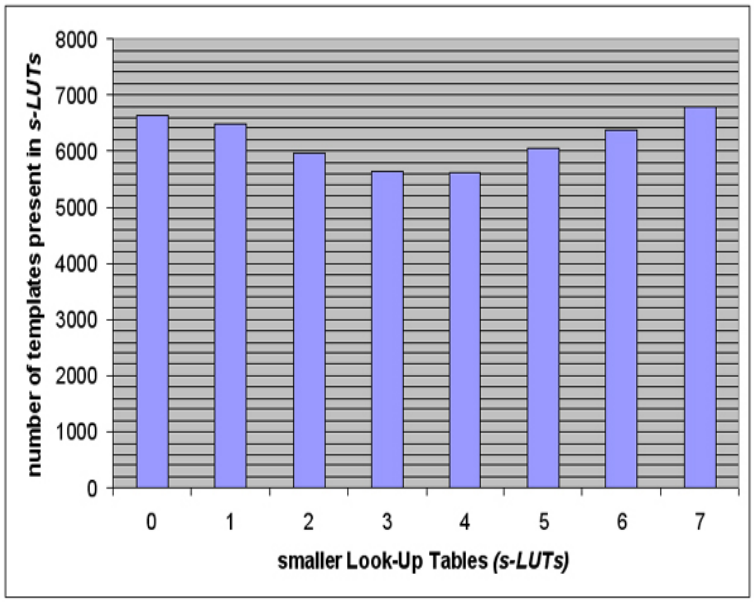

Figure 4.4: Graph showing partitioning of templates to $s-LUTs$ when Only Addition (OA) function with $N = 8$ is used.

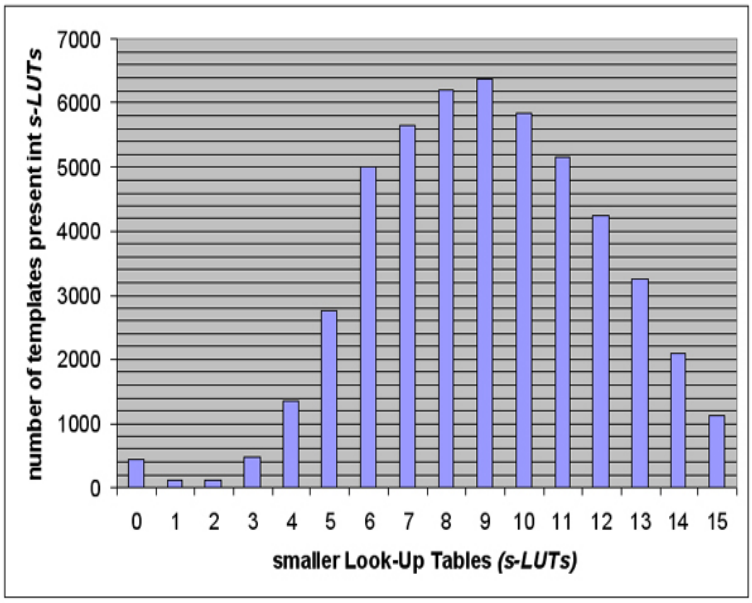

Figure 4.5: Graph showing partitioning of templates to $s-LUTs$ when Only Addition (OA) function with $N = 16$ is used.

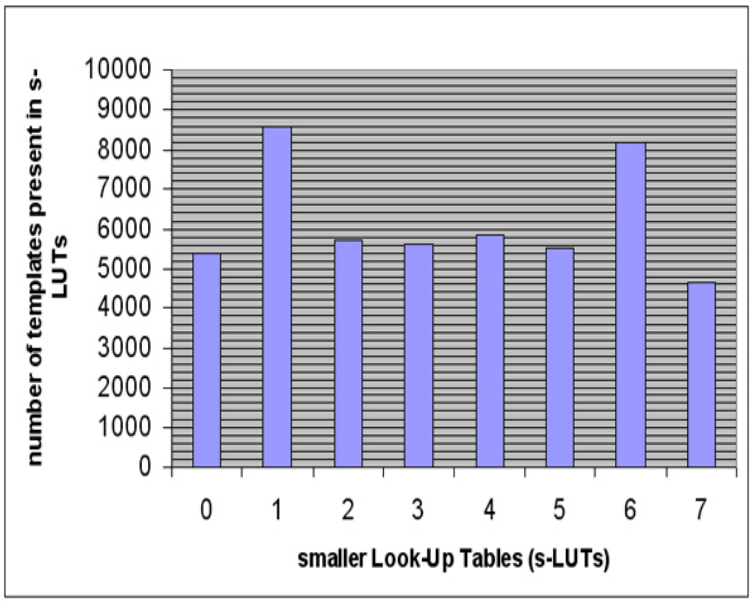

Figure 4.6: Graph showing partitioning of templates to $s - LUTs$ when Mod Only (MO) function with $N = 8$ is used.

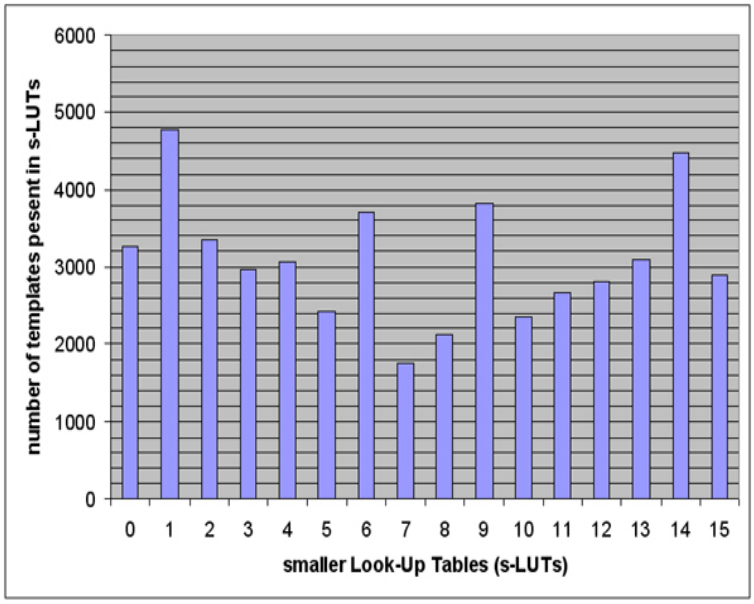

Figure 4.7: Graph showing partitioning of templates to $s - LUTs$ when Mod Only (MO) function with $N = 16$ is used.

4.2 Simulation of Parallel Inverse Halftoning and Discussion

The results and images obtained from proposed algorithms are shown and discussed in this section.

4.2.1 XM with Pixel Loss (XM_PL) Algorithm

To execute XM_PL algorithm the toolbox finds values of m or $mean$ with the generation of LUT and stores it in a public static variable of an external class. The steps taken place in order to simulate the proposed XM_PL algorithm are shown in Figure 4.8 The test set consists of images *peppers*, *clock*, and *boat*. The results shows an average of results obtained from test set. The toolbox also takes input for number of concurrently fetched templates (k) and number of smaller Look-Up Tables ($s - LUT$)

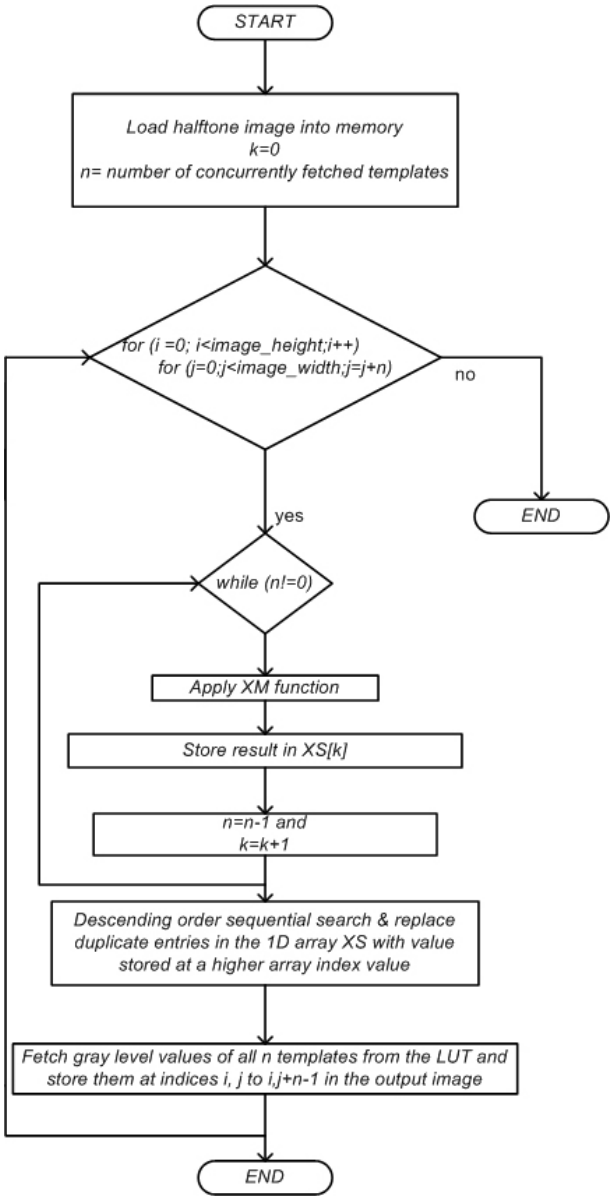

Figure 4.8: Flowchart showing steps of execution to simulate Xor with Mean with Pixel Loss (XM_PL) algorithm.

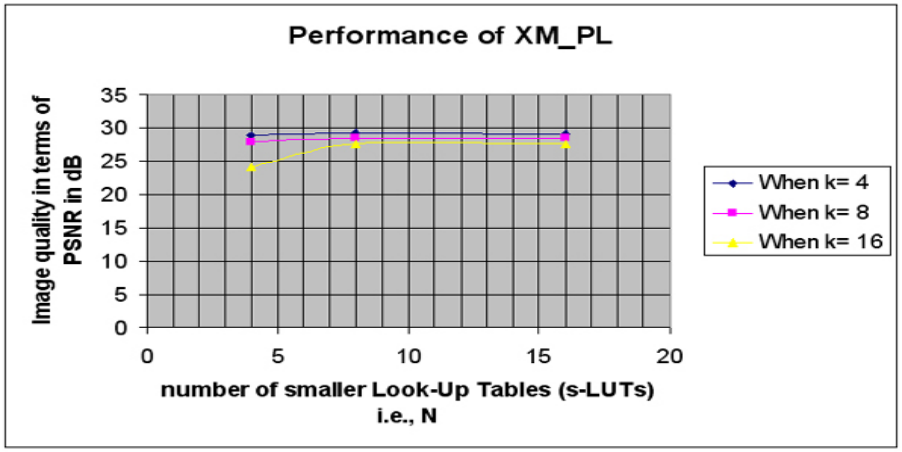

Figure 4.9: Plot showing performance of Xor with Mean with Pixel Loss (XM_PL) algorithm in terms image quality versus N for different values of k.

(N). The results obtained are summarized in graph shown in Figure 4.9. The graph shows that we have best image quality when N= 16 for any value of k from 2 to 20. However, N=16 results into high I/O (Input/Output) pin count on target VLSI Implementation. If I/O pin count is not a constraint on target VLSI platform then we can use N= 16 otherwise we use N= 8. The total entries in all $s-LUTs$ when N= 16 is equal to the total entries when N =8. Four original continuous tone images, images obtained from serial LUT method, and images obtained from proposed algorithm are shown in Figure 4.10 to Figure 4.24 when N=8 and k= 4.

4.2.2 XM with No Pixel Loss (XM_NPL) Algorithm

To execute XM_NPL algorithm the toolbox finds value of m or *mean* with the generation of LUT and stores it in a public static variable of an external class. The steps required in order to simulate the XM_NPL algorithm are shown in Figure 4.25. The gain in clock cycles over the serial LUT method is measured by counting the maximum

Figure 4.10: Original continuous tone image named Peppers.

Figure 4.11: Inverse halftoned images obtained from serial LUT method (PSNR= 24.4154 dB).

Figure 4.12: Inverse halftoned image obtained from XM_PL algorithm (PSNR= 29.2605 dB).

Figure 4.13: Original continuous tone image named Trees.

Figure 4.14: Inverse halftoned images obtained from serial LUT method (PSNR= 27.9463 dB).

Figure 4.15: Inverse halftoned image obtained from XM_PL algorithm (PSNR= 27.6723 dB).

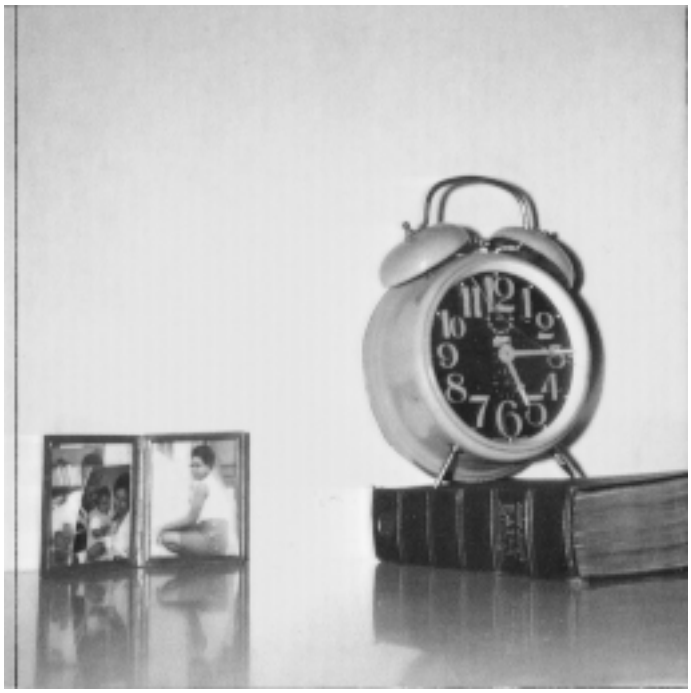

Figure 4.16: Original continuous tone image named Clock.

Figure 4.17: Inverse halftoned images obtained from serial LUT method (PSNR= 30.1680 dB).

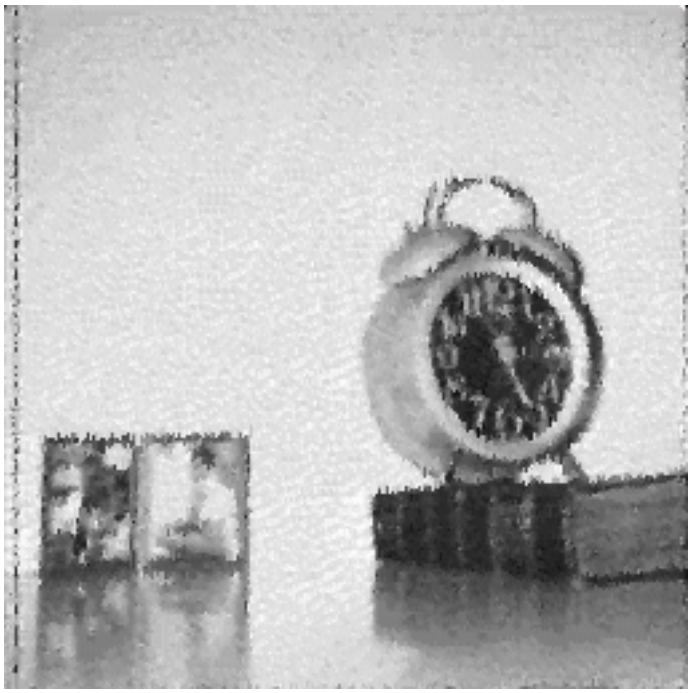

Figure 4.18: Inverse halftoned image obtained from XM_PL algorithm (PSNR= 30.0847 dB).

Figure 4.19: Original continuous tone image named Boat.

Figure 4.20: Inverse halftoned image obtained from serial LUT method (PSNR= 30.1861 dB).

Figure 4.21: Inverse halftoned image obtained from XM_PL algorithm (PSNR= 28.5448 dB).

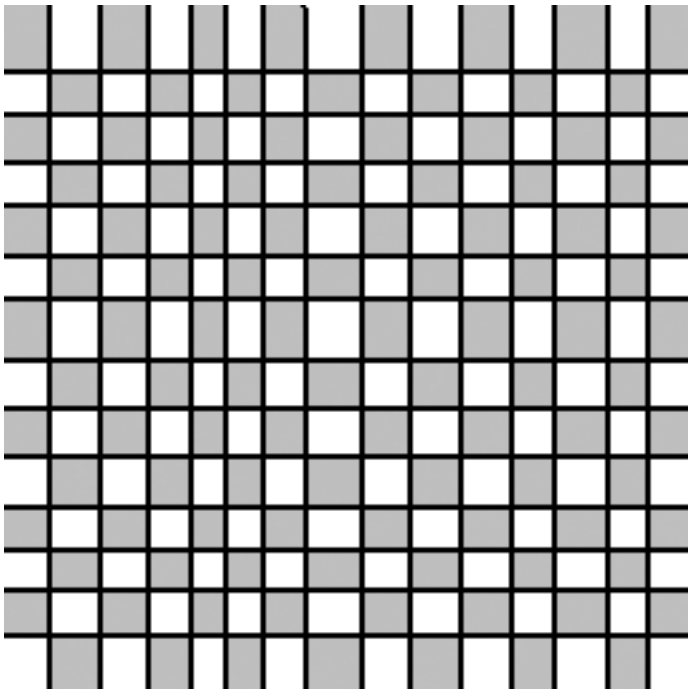

Figure 4.22: Original continuous tone image named Squares.

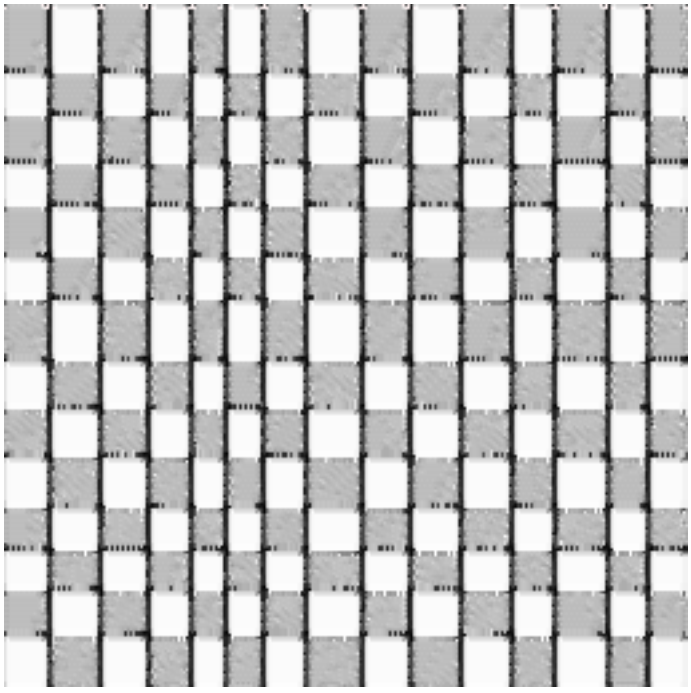

Figure 4.23: Inverse halftoned image obtained from serial LUT method (PSNR= 17.2832 dB)

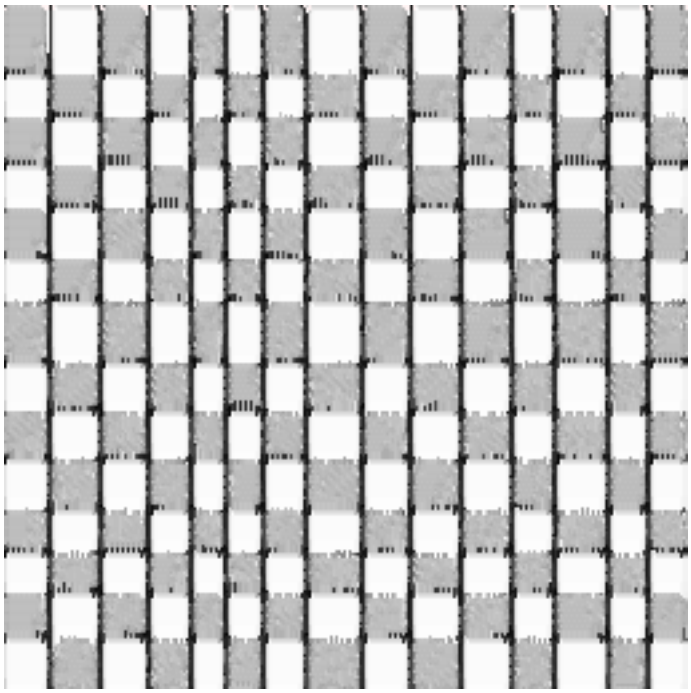

Figure 4.24: Inverse halftoned image obtained from XM_PL algorithm (PSNR= 14.9323 dB).

number of duplicate values in each iteration. This gives the number of clock cycles the pipeline should be stalled for, in order, that all templates in queues fetch their *contone* values. The box sequential Search and Replace measures the duplicate results from XM functions and measure of occurrences of any one duplicate value. This gives the pipeline stalls for that fetch cycle. The test images consists of *peppers*, *clock*, and *boat* and an average of results obtained from these images are shown here. The tool-box also takes input for *number of concurrently fetched templates* = k and *number of smaller Look-Up Tables* ($s - LUT$)=N. The results obtained are summarized in graph shown in Figure 4.26. The graph shows the number of cycles that can be saved using the proposed algorithm over the serial LUT method of inverse halftoning versus the number of concurrently fetched templates (k) and for given values of N. The output

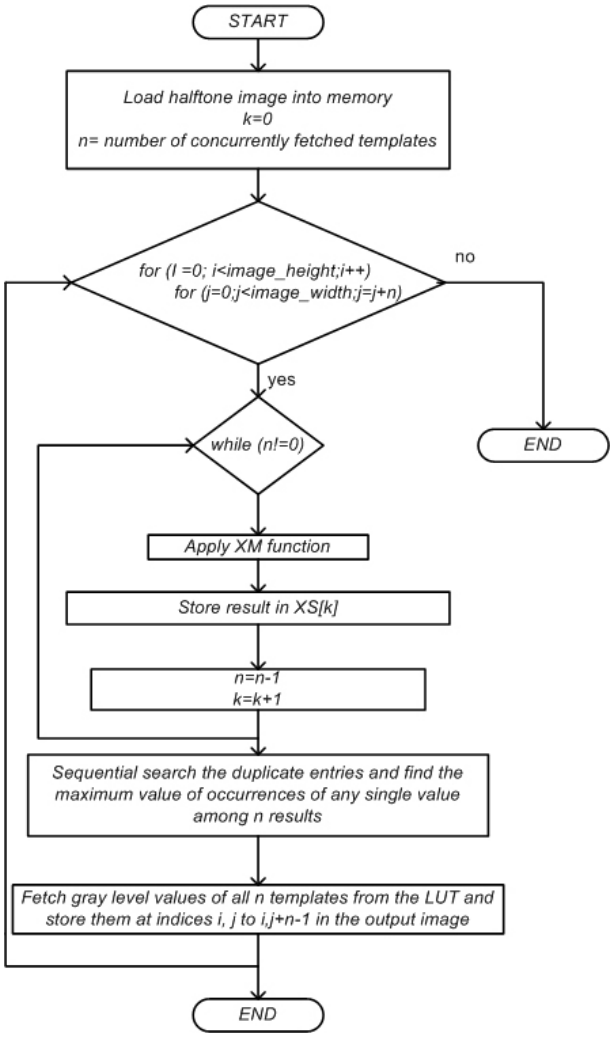

Figure 4.25: Flowchart showing steps of execution to simulate XM_NPL algorithm.

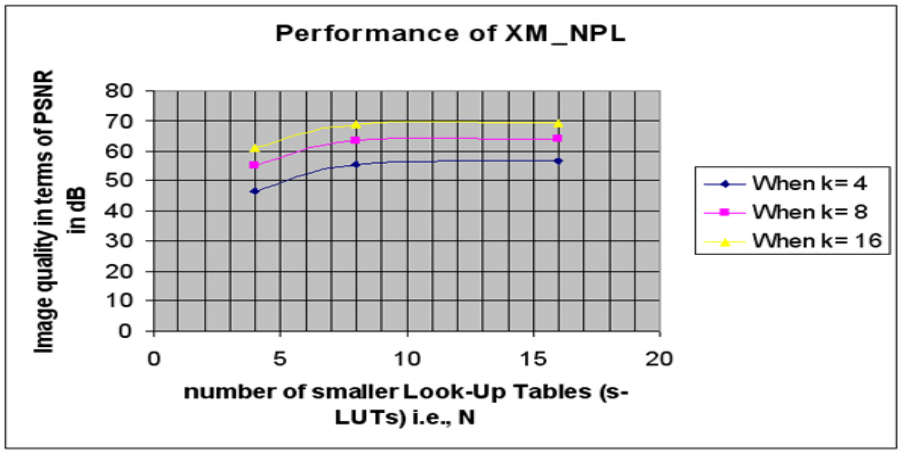

Figure 4.26: Plot showing performance of Xor with Mean with No Pixel Loss (XM_NPL) algorithm in terms image quality versus N for different values of k.

images are same as obtained by serial LUT method and are therefore not shown. The graph shows that when $N = 16$ i.e., when number of smaller Look-Up Tables ($s - LUT$) is equal to 16 we have the maximum gain in clock cycles over the serial LUT method. But since $N = 16$ takes more I/O pins on target VLSI implementation we prefer to use $N = 8$ that has 50% less I/O count for having 50% outputs than $N = 16$. However, if I/O pin count is not a limitation then we should use $N = 16$. The number of LUT entries in $N = 16$ is equal to number of entries in $N = 8$. We also notice that when value of k increases the gain in clock cycles also increases but as a result we need large size queues before smaller Look-Up Tables ($s - LUT$) because queues before $s - LUT$ must be of size equal to k.

4.2.3 OA with Pixel Loss (OA_PL) Algorithm

The toolbox perform inverse halftoning using OA_PL algorithm when it is selected in the GUI. The procedure is similar to XM_PL except that OA function is used in

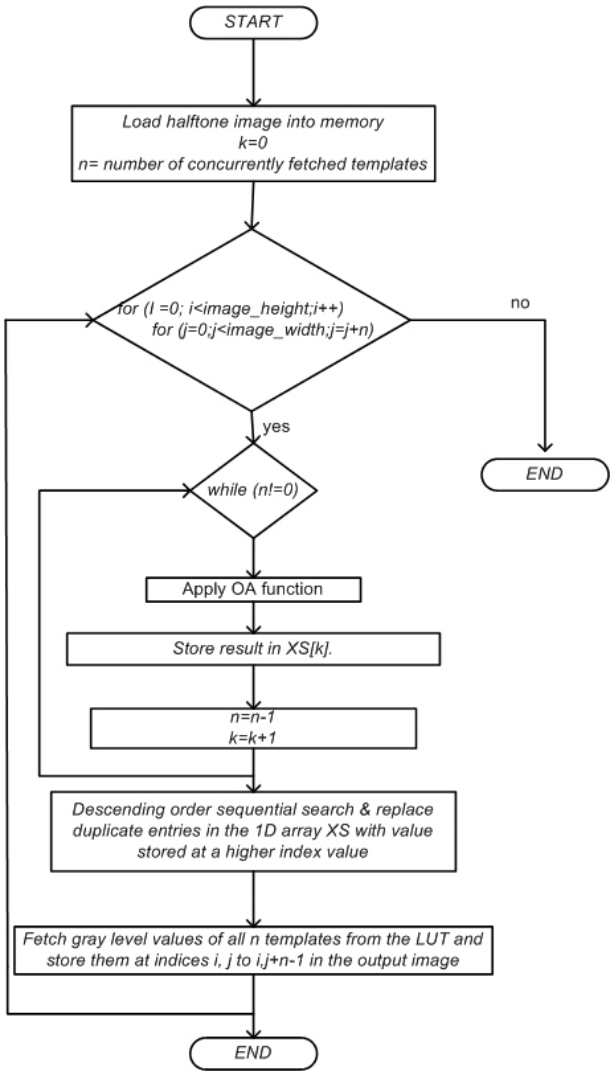

Figure 4.27: Flowchart showing steps of execution to simulate Only Addition with Pixel Loss (OA_PL) algorithm.

place of XM function. The steps taken place in order to execute OA_PL algorithm and display resulting images and calculate image quality in terms of $PSNR$ is shown in Figure 4.27. The discarding of some templates as occurs is OA_PL algorithm is performed in box "Sequential Search and Replace". The test set input to the toolbox consists of images: peppers, clock, and boat. The toolbox also takes input for N and k. The results obtained from the algorithm are shown using graph in Figure 4.28. Along $y - axis$, the image quality in terms of $PSNR$ is shown and along $x - axis$,

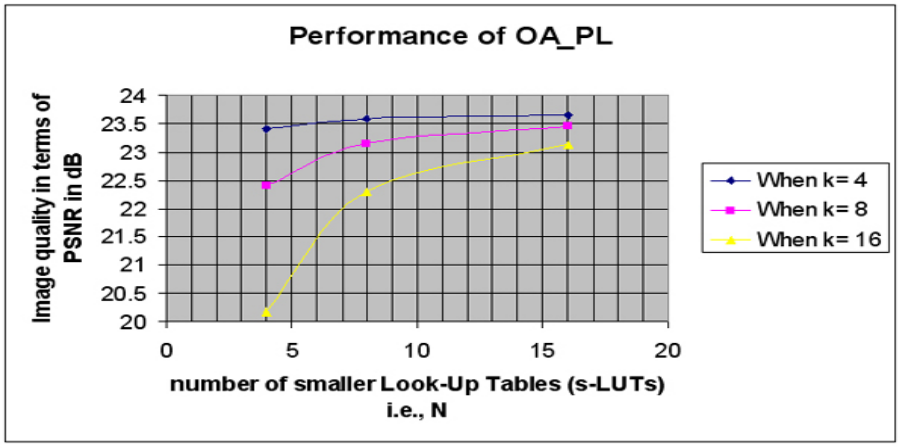

Figure 4.28: Plot showing performance of Only Addition with Pixel Loss (OA_PL) algorithm in terms image quality versus N for different values of k.

the number of concurrently fetched templates i.e., k is shown. Each line in the graph refers to different value of N. It can be inferred from the graph that $N = 16$ has best image quality and then comes $N = 8$. However $N = 8$ have less I/O pins as compared to $N = 16$. Some images obtained from the algorithm with parameters $k = 4$ and $N = 8$ are shown in Figures 4.31 to 4.33.

4.2.4 OA with No Pixel Loss (OA_NPL) Algorithm

The toolbox can execute OA_NPL algorithm when the option corresponding to it is selected in the GUI. The execution is similar to XM_NPL algorithm and the results are also displayed in the same way as for XM_NPL algorithm. The steps to execute the algorithm and measure performance parameters are shown in Figure 4.34. The results obtained are illustrated using graph in Figure 4.35. The graph shows that when N= *16* i.e., when number of smaller Look-Up Tables ($s - LUT$) is equal to 16 it shows maximum gain in clock cycles over the serial LUT method. But $N = 16$

Figure 4.29: Inverse halftoned image Obtained from Only Addition with Pixel Loss (OA_PL) algorithm (PSNR= 24.5309 dB).

Figure 4.30: Inverse halftoned image Obtained from OA_PL algorithm (PSNR= 21.9254 dB).

Figure 4.31: Inverse halftoned image Obtained from OA_PL algorithm (PSNR= 24.2860 dB).

Figure 4.32: Inverse halftoned image obtained from OA_PL (PSNR= 22.9444 dB).

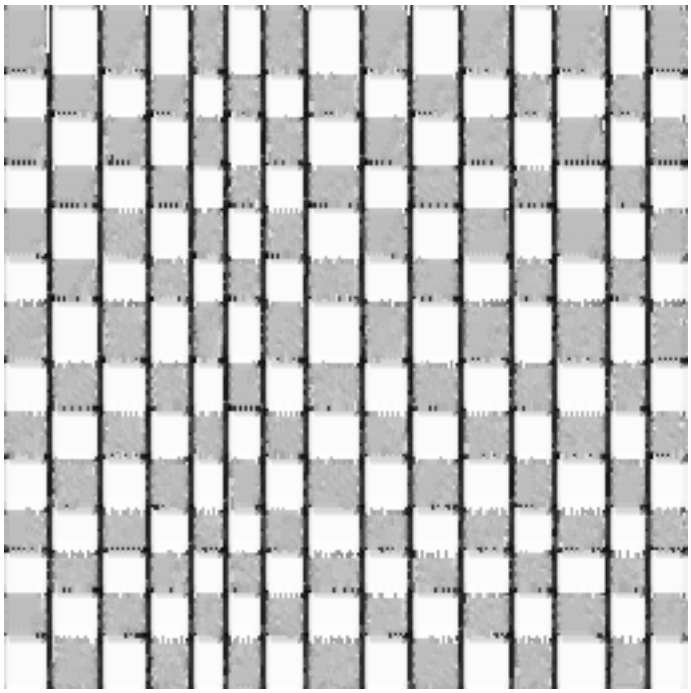

Figure 4.33: Inverse halftoned image obtained from OA_PL algorithm (PSNR= 14.3188 dB)

takes more I/O pins on target VLSI implementation than $N = 8$. However, if I/O pin count is not a limitation then one should use $N = 16$. The number of LUT entries in $N = 16$ is equal to number of entries in $N = 8$. It is also shown that with the increase of k gain in clock cycles also increases. High k values also increases size of queues before $s - LUTs$ however, they do not increase pin count considerably.

4.2.5 MO with Pixel Loss (MO_PL) Algorithm

The MO_PL algorithm is also executed and same set of images are inverse halftoned. The simulation follows the same steps as followed in XM_PL and OA_PL algorithms. Few images obtained from the proposed algorithm are shown in Figures 4.37, 4.38, and 4.39. The performance of MO_PL algorithm in terms of image quality represented by

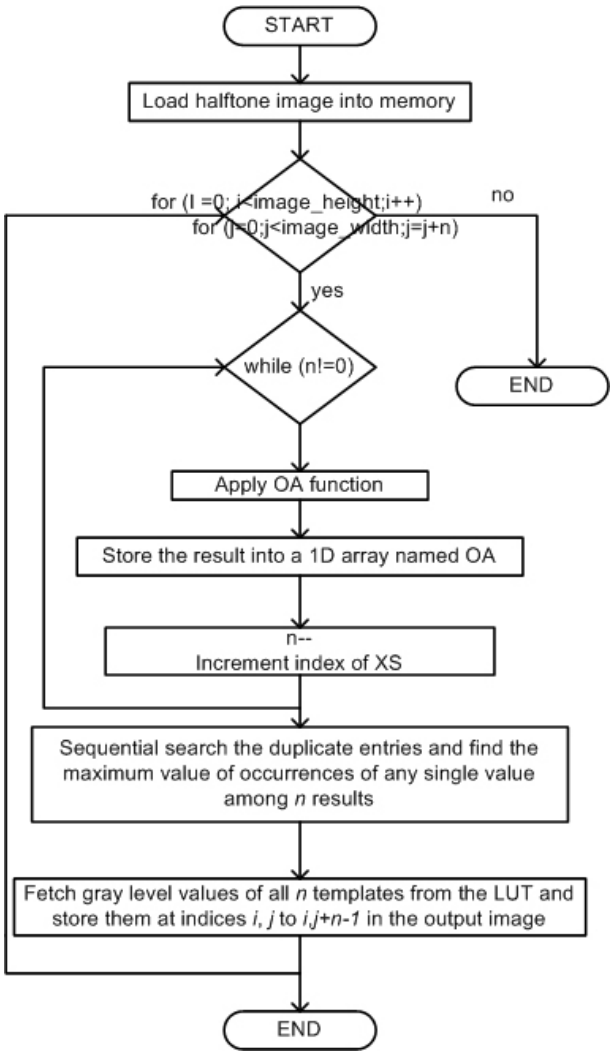

Figure 4.34: Flowchart showing steps of execution to simulate Only Addition with No Pixel Loss (OA_NPL) algorithm.

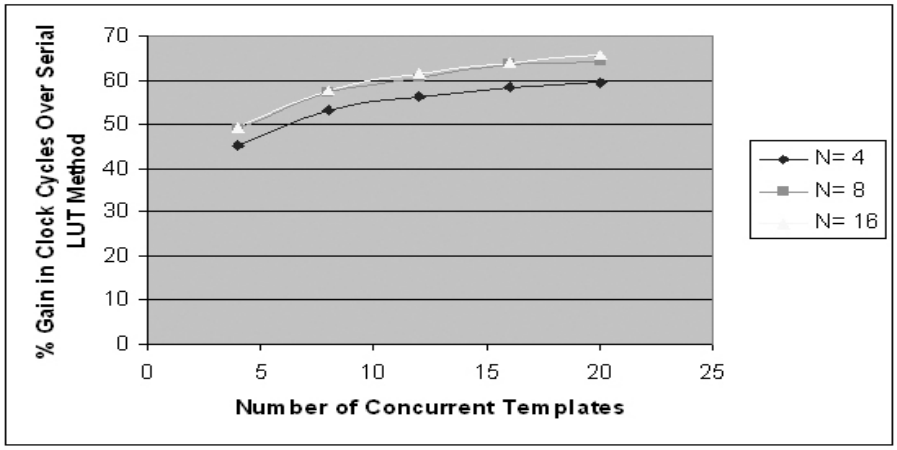

Figure 4.35: Plot showing performance of Only Addition with No Pixel Loss (OA_NPL) algorithm in terms image quality versus N for different values of k.

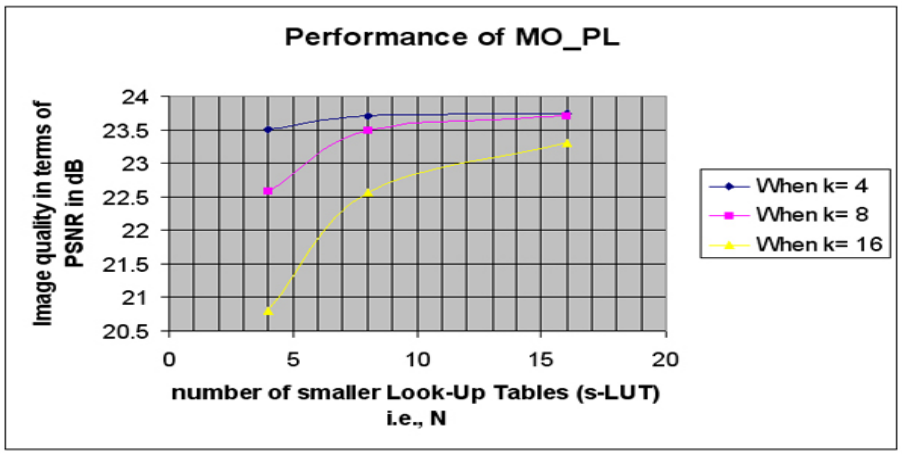

Figure 4.36: Plot showing performance of Mod Only with Pixel Loss (MO_PL) algorithm in terms image quality versus N for different values of k.

Peak Signal to Noise Ratio ($PSNR$) versus the number of concurrently fetched templates (k) and number of smaller Look-Up Tables ($s-LUTs$) is shown in Figure 4.36. Sample images obtained from the algorithm are also shown in Figures 4.37, 4.38, 4.39, and 4.40.

Figure 4.37: Inverse halftoned image Obtained from MO_PL algorithm (PSNR= 24.6200 dB).

Figure 4.38: Inverse halftoned image Obtained from MO_PL algorithm (PSNR= 22.1285 dB).

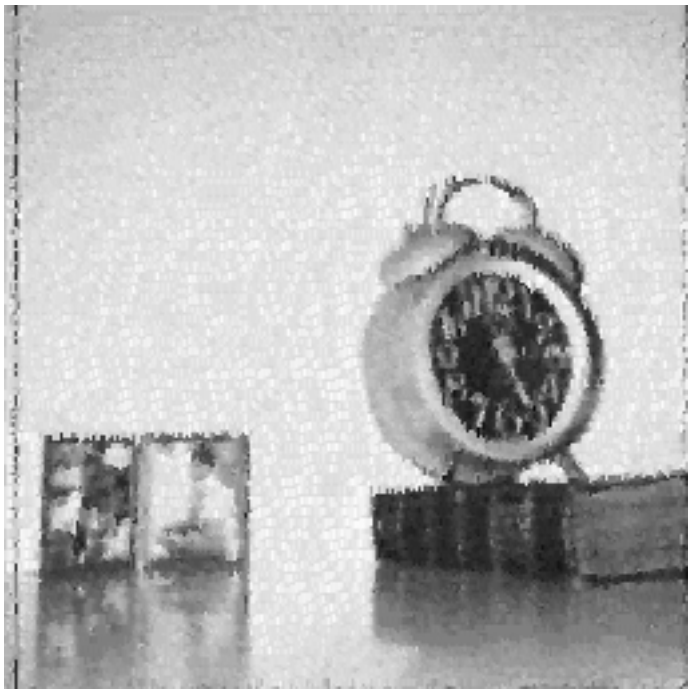

Figure 4.39: Inverse halftoned image Obtained from MO_PL algorithm (PSNR= 24.3663 dB).

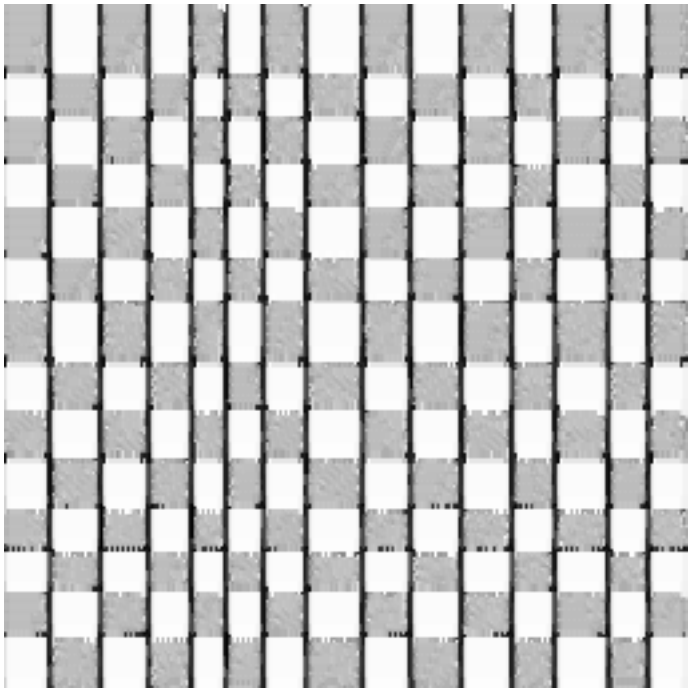

Figure 4.40: Inverse halftone image obtained from MO_PL algorithm (PSNR= 14.4180 dB).

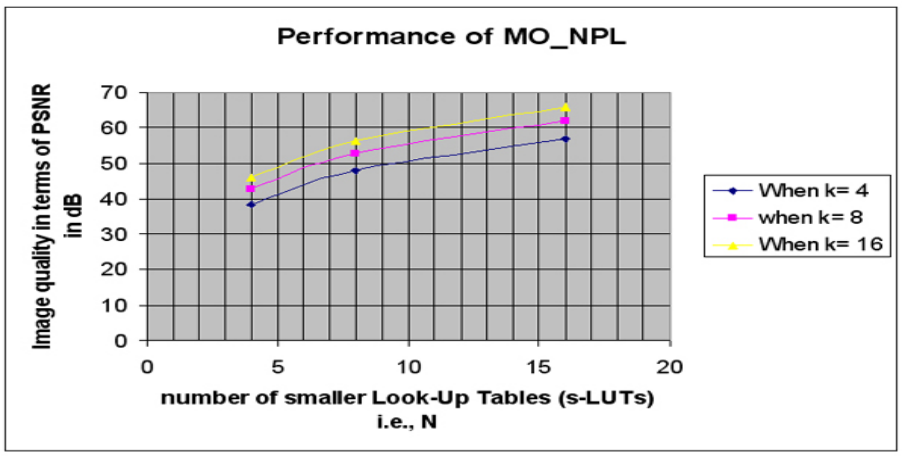

Figure 4.41: Plot showing performance of Mod Only with No Pixel Loss (MO_NPL) algorithm in terms image quality versus N for different values of k.

4.2.6 MO with No Pixel Loss (MO_NPL) Algorithm

The algorithm MO_NPL is also executed and its performance is measured in terms of percentage of gain in clock cycles it can provide over the serial LUT method. The image quality is also measured however, it remains similar to the serial LUT method and no change in image quality occurs with parallelization, therefore it is not shown again. The results are shown using graph in Figure 4.41 in which $x-axis$ shows the number of smaller Look-Up Tables ($s-LUTs$) and $y-axis$ shows the *percentage gain over serial LUT method*.

4.3 Comparison

A comparison of all performance graphs reveal that Xor with Mean with Pixel Loss (XM_PL), Only Addition with Pixel Loss (OA_PL), and Mod Only with Pixel Loss (MO_PL) algorithms show same behavior i.e., all algorithms have good image quality when value of k is small and value of N is large. However, at the same time there is not

much improvement in image quality when value of N changes from 8 to 16 as compared to when N values changes from 4 to 8. The graphs comparison also reveal that performance of Xor with Mean with No Pixel Loss (XM_NPL), Only Addition with No Pixel Loss (OA_NPL), and Mod Only with No Pixel Loss (MO_NPL) algorithms in terms of gain in clock cycles over the serial LUT method improves as the value k and N increases.

4.4 Simulation of RXC_PL Algorithm

The RXC_PL algorithm has the disadvantage that it may result into very large $s-LUTs$ sizes and as a result the benefit of parallelization becomes reduced. This problem is not present with other proposed algorithms in which the number of entries in $s-LUTs$ remains equal to the entries in the LUT of serial method and therefore they offer high saving in area. This section shows the simulation of RXC_PL algorithm. It is assumed that inverse halftone operation is ideal and consequently, the results show quality degradation that occurs due to Pixel Loss in discarding templates in the algorithm. There is no quality loss shown due to inverse halftone operation in the resulting images. The results are summarized in Table 4.1. They have high PSNR values because inverse halftoning is considered ideal. The column $Halftone\_Algorithm$ shows the name of halftone algorithm that is used in the halftone images. In Table 4.1 terms, "FS ED" refers to Floyd and Steinberg Error Diffusion method [10], "GN ED" refers Green Noise Error Diffusion method [23], "EG ED" refers to Edge Enhancement Error Diffusion method [24].

Table 4.1: Results of RXC_PL algorithm.

Image	Halftone Algorithm	Percentage of templates discarded	PSNR of output images
Boat	FS ED [10]	65.0864	30.3749
Clock	FS ED [10]	70.6667	30.1671
Lena	FS ED [10]	70.9629	28.7139
Boat	GN ED [23]	63.7531	31.2554
Clock	GN ED [23]	69.8765	31.7895
Lena	GN ED [23]	69.7284	29.5628
Boat	EG ED [24]	67.3086	32.1370
Clock	EG ED [24]	68.5926	29.9289
Lena	EG ED [24]	71.0617	28.2293

4.5 Summary

In this chapter first smaller Look-Up Tables ($s - LUTs$) are generated using the proposed algorithms and each algorithm is evaluated in its ability to generate $s-LUTs$ of equal sizes. Then the performance of proposed algorithms in terms of image quality and $PSNR$ (Peak Signal to Noise Ratio) and gain in clock cycles over the serial LUT method is shown. XM_PL and OA_PL algorithms have constant gain in clock cycles and their image quality varies. It is observed that XM_PL have better image quality than OA_PL and MO_PL and the quality is high for higher values of N for any number of concurrently fetched templates. Similarly XM_NPL, OA_NPL and MO_NPL algorithms have constant image quality that is same to serial LUT method. The gain in clock cycles over the serial LUT method is more for XM_PL algorithm than OA_PL and MO_NPL algorithms. The images obtained from each algorithm are also shown to show the visual quality of images. RXC_PL is simulated however it cannot guarantee saving in LUT size over serial LUT method.

CHAPTER 5

HARDWARE DESIGN AND MODELING

This chapter presents the hardware design and modeling of proposed algorithms. The hardware models are completely synthesizable and show the approximated hardware area occupied by the algorithms and maximum clock frequencies achievable on target FPGAs. VHDL language is used to model the computational part of the algorithms and smaller Look-Up Tables ($s - LUT$) are assumed to reside on external memories. The designs are pipelined and divided into stages. Each stage can execute in parallel on inputs from its predecessor stage. The pipelining also allows fetching templates on every clock cycle from the halftone image. The target platform is Xilinx Spartan 3E FPGA that has $100K$ gates and up to 144 I/O ports. The design statistics are obtained using Xilinx ISE synthesizer and "place and route" tool. The performance of the hardware model is discussed with respect to the following three features:

1. Smaller pipeline stages for maximum frequency,

2. Resource utilization on target FPGA platforms.

The hardware design and model of each algorithm is described in the rest of the chapter.

5.1 Xor with Mean with Pixel Loss (XM_PL) Algorithm

The integrated circuit that implements the XM_PL algorithm with parameters *k=4 and N=8* consists of the eight blocks and each block is separated from the other block by pipeline registers. In this way the design is pipelined and each block can operate concurrently on different inputs. New templates can be also fetched from the halftone image on every clock cycle. A block diagram is shown in Figure 5.1 in which the computational blocks are shown interleaved with pipelined registers. The detail of computation in blocks is shown in the rest of this section.

5.1.1 Block 1

In this stage four templates I_0, I_1, I_2, and I_3 are fetched from the halftone image and stored in registers t_0, t_1, t_2, and t_3 respectively. The Boolean equations representing operations in this stage are as follows:

$t_0(0 \cdots p-1) \leftarrow I_0(0 \cdots p-1)$,

$t_1(0 \cdots p-1) \leftarrow I_1(0 \cdots p-1)$,

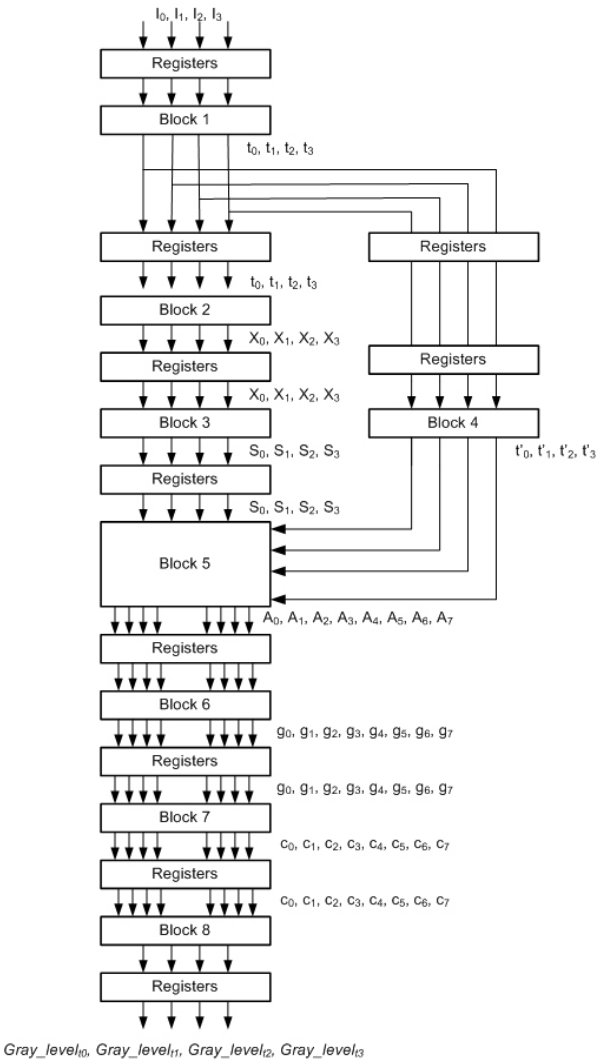

Figure 5.1: Illustration showing block diagram of the hardware model of proposed algorithms.

$t_2(0 \cdots p-1) \leftarrow I_2(0 \cdots p-1)$,

$t_3(0 \cdots p-1) \leftarrow I_3(0 \cdots p-1)$.

5.1.2 Block 2

In this block four templates ($t_0,\ t_1,\ t_2,\ and\ t_3$) go through XOR operation with m. The operations in this block are represented by following equations:

$X_i(0 \cdots p-1) \leftarrow t_i(0 \cdots p-1)\ XOR\ m(0 \cdots p-1)$

where i=0, 1, 2, and 3.

5.1.3 Block 3

In this block XOR results are added to find the number of ones present in them. It is performed using a Carry Save Adder (CSA) tree for each XOR result separately. The CSA trees have 15 $CSAs$ when template type is 19$opts$. It is illustrated in Figure 5.2. The equations representing computation in this block are shown below:

$S_i(0 \cdots 2) \leftarrow CSA\_TREE_i(X_i(0 \cdots p-1))$

Where i=0, 1, 2, 3.

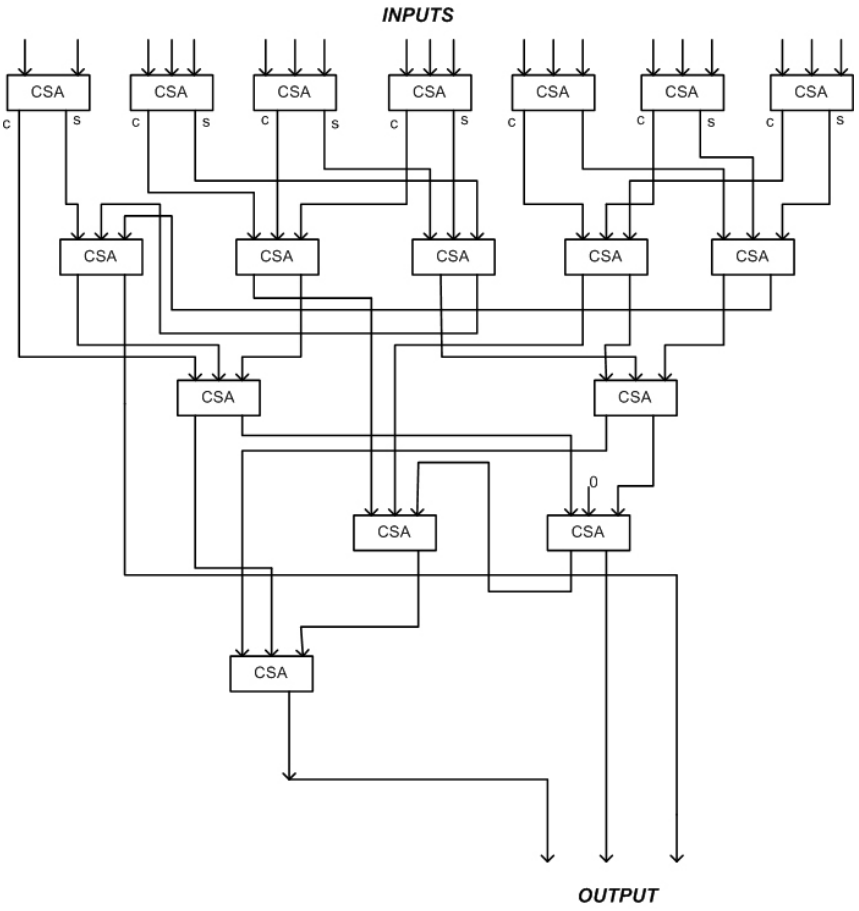

Figure 5.2: Carry Save Adder (CSA) Tree.

5.1.4 Block 4

In this block templates t_0, t_1, t_2, and t_3 are appended with sequence numbers 001, 010, 011, and 100 respectively. The Boolean expressions representing operations in this block are:

$t'_0(0 \cdots p+2) \leftarrow t_0(0 \cdots p-1)$ & 001

$t'_1(0 \cdots p+2) \leftarrow t_1(0 \cdots p-1)$ & 010

$t'_2(0 \cdots p+2) \leftarrow t_2(0 \cdots p-1)$ & 011

$t'_3(0 \cdots p+2) \leftarrow t_3(0 \cdots p-1)$ & 100

5.1.5 Block 5

This block consists of four 1x8 demultiplexers. The Boolean equations that represent the logic in this stage are:

$A_i[0](0 \cdots p-1) \leftarrow \overline{slut_i(2)} \cdot \overline{slut_i(1)} \cdot \overline{slut_i(0)} \cdot (t'_i(o \cdots p+2))$

$A_i[1](0 \cdots p-1) \leftarrow \overline{slut_i(2)} \cdot \overline{slut_i(1)} \cdot slut_i(0) \cdot (t'_i(o \cdots p+2))$

$A_i[2](0 \cdots p-1) \leftarrow \overline{slut_i(2)} \cdot slut_i(1) \cdot \overline{slut_i(0)} \cdot (t'_i(o \cdots p+2))$

$A_i[3](0 \cdots p-1) \leftarrow \overline{slut_i(2)} \cdot slut_i(1) \cdot slut_i(0) \cdot (t'_i(o \cdots p+2))$

$A_i[4](0 \cdots p-1) \leftarrow slut_i(2) \cdot \overline{slut_i(1)} \cdot \overline{slut_i(0)} \cdot (t'_i(o \cdots p+2))$

$A_i[5](0 \cdots p-1) \leftarrow slut_i(2) \cdot \overline{slut_i(1)} \cdot slut_i(0) \cdot (t'_i(o \cdots p+2))$

$A_i[6](0 \cdots p-1) \leftarrow slut_i(2) \cdot slut_i(1) \cdot \overline{slut_i(0)} \cdot (t'_i(o \cdots p+2))$

$A_i[7](0 \cdots p-1) \leftarrow slut_i(2) \cdot slut_i(1) \cdot slut_i(0) \cdot (t'_i(0 \cdots p+2))$

where i=0 for first demultiplexer, i=1 for second demultiplexer and so on. The

variables $A_i[0]$ to $A_i[7]$ represents eight outputs from i^{th} demultiplexer.

5.1.6 Block 6

This Block consists of eight 8x1 multiplexers. The Boolean equations representing logic in this stage in terms of previously computed variables are shown below:

$$g_i(0 \cdots p+2) \leftarrow (A_3[i](p) + A_3[i](p+1) + A_3[i](p+2)) \cdots A_3[i](0 \cdots p+2) + \overline{(A_3[i](p) + A_3[i](p+1) + A_3[i](p+2))} \cdot (A_2[i](p) + A_2[i](p+1) + A_2[i](p+2)) \cdot A_2[i](0 \cdots p+2) + \overline{(A_3[i](p) + A_3[i](p+1) + A_3[i](p+2))} \cdot \overline{(A_2[i](p) + A_2[i](p+1) + A_2[i](p+2))} \cdot (A_1[i](p) + A_1[i](p+1) + A_1[i](p+2)) \cdot (A_1[i](0 \cdots p+2)) + \overline{(A_3[i](p) + A_3[i](p+1) + A_3[i](p+2))} \cdot \overline{(A_2[i](p) + A_2[i](p+1) + A_2[i](p+2))} + \overline{(A_1[i](p) + A_1[i](p+1) + A_1[i](p+2))} \cdot (A_0[i](p) + A_0[i](p+1) + A_0[i](p+2)) \cdot A_0[i](0 \cdots p+2)$$

where i=0 to 7 and i=0 refers to first multiplexer, i=1 refers to second multiplexer, and so on. The output g_0 is from the first multiplexer, g_1 is from the second multiplexer, and so on.

5.1.7 Block 7

This block contains implementations of smaller Look-Up Tables ($s-LUT$) using Content Addressable Memory (CAM) and Read Only Memory (ROM) pairs. The block diagram in Figure 5.3 shows implementation of one $s-LUT$. CAM stores

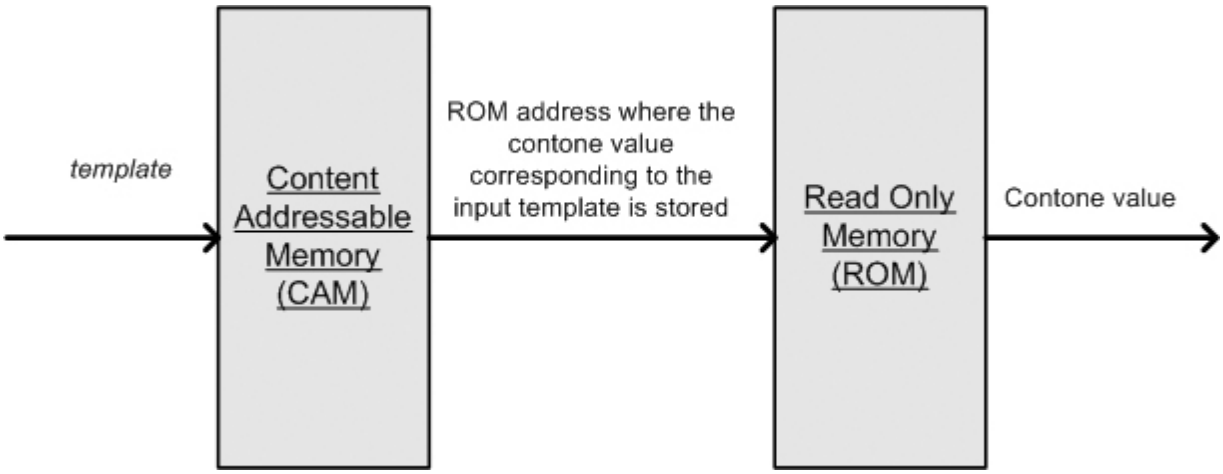

Figure 5.3: Implementation of smaller Look-Up Table ($s - LUT$) using CAM-ROM.

templates that are assigned to that $s - LUT$ and it returns address of the adjacent ROM where the corresponding contone values are stored. Upon inputting of address ROM returns the contone value. The Boolean equation below show operations performed in this stage:

$number\ of\ entries\ in\ the\ sLUT = 2^d - 1$

$number\ of\ gray\ levels = \ 256$

$x_i(0 \cdots d-1) \leftarrow CAM_i(g_i(0 \cdots p-1))$

$c_i(0 \cdots 7) \leftarrow ROM_i(x_i(\cdots d-1)$

where i=0, 1, 2, 3, 4, 5, 6, and 7.

This $CAM - ROM$ should always be used when Look-Up Table stores contone values of templates less than $2^{(width\ of\ the\ CAM)}$. In this way the LUT of serial LUT method is also implemented using the same approach.

$a_i \leftarrow \overline{g_i(p)} \cdot \overline{g_i(p+1)} \cdot g_i(p+2)$
Where i=0 to 7. $a_8(0 \cdots 7) \leftarrow a_0 \cdot c_0(0 \cdots 7) + a_1 \cdot c_1(0 \cdots 7)$
$+a_2 \cdot c_2(0 \cdots 7) + a_3 \cdot c_3(0 \cdots 7) + a_4 \cdot c_4(0 \cdots 7)$
$+a_5 \cdot c_5(0 \cdots 7) + a_6 \cdot c_6(0 \cdots 7) + a_7 \cdot c_7(0 \cdots 7)$
$a_9 \leftarrow a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7$
$a_{10}(0 \cdots 7) \leftarrow a_9 \cdot a_8(0 \cdots 7) + \overline{a_9} \cdot a_8(0 \cdots 7)$
$Gray\_level_{t_0}(0 \cdots 7) \leftarrow a_{10}(0 \cdots 7)$
Where $Gray\_level_{t_0}$ is the contone value of the template t_0.

Figure 5.4: Circuit after $s - LUTs$ to output contone value of template t_0.

$b_i \leftarrow \overline{g_i(p)} \cdot g_i(p+1) \cdot \overline{g_i(p+2)}$
Where i= 0 to 7. $b_8(0 \cdots 7) \leftarrow b_0 \cdot c_0(0 \cdots 7) + b_1 \cdot c_1(0 \cdots 7)$
$+b_2 \cdot c_2(0 \cdots 7) + b_3 \cdot c_3(0 \cdots 7) + b_4 \cdot c_4(0 \cdots 7)$
$+b_5 \cdot c_5(0 \cdots 7) + b_6 \cdot c_6(0 \cdots 7) + b_7 \cdot c_7(0 \cdots 7)$
$b_9 \leftarrow b_0 + b_1 + b_2 + b_3 + b_4 + b_5 + b_6 + b_7$
$b_{10}(0 \cdots 7) \leftarrow b_9 \cdot b_8(0 \cdots 7) + \overline{b_9} \cdot b_8(0 \cdots 7)$
$Gray\_level_{t_1}(0 \cdots 7) \leftarrow b_{10}(0 \cdots 7)$
Where $Gray\_level_{t_1}$ is the contone value of the template t_1.

Figure 5.5: Circuit after $s - LUTs$ to output contone value of template t_1.

5.1.8 Block 8

At multiplexers in Block 6 only one input was selected, however, more than one can be high at same time. Therefore, several templates are discarded. This stage assigns them contone values from neighboring non-discarded templates. The operations are illustrated with the help of Boolean equations in Figure 5.4 to Figure 5.7, in which neighboring contone values are used in place of actual contone value if the corresponding template is discarded.

$d_i \leftarrow \overline{g_i(p)} \cdot g_i(p+1) \cdot g_i(p+2)$
Where i= 0 to 7. $d_8(0 \cdots 7) \leftarrow a_0 \cdot c_0(0 \cdots 7) + d_1 \cdot c_1(0 \cdots 7)$
$+d_2 \cdot c_2(0 \cdots 7) + d_3 \cdot c_3(0 \cdots 7) + d_4 \cdot c_4(0 \cdots 7)$
$+d_5 \cdot c_5(0 \cdots 7) + d_6 \cdot c_6(0 \cdots 7) + d_7 \cdot c_7(0 \cdots 7)$
$d_9 \leftarrow d_0 + d_1 + d_2 + d_3 + d_4 + d_5 + d_6 + d_7$
$d_{10}(0 \cdots 7) \leftarrow d_9 \cdot d_8(0 \cdots 7) + \overline{d_9} \cdot d_8(0 \cdots 7)$
$Gray\_level_{t_2}(0 \cdots 7) \leftarrow d_{10}(0 \cdots 7)$
Where $Gray\_level_{t_2}$ is the contone value of the template t_2.

Figure 5.6: Circuit after $s - LUTs$ to output contone value of template t_2.

$e_i \leftarrow \overline{g_i(p)} \cdot \overline{g_i(p+1)} \cdot g_i(p+2)$
Where i= 0 to 7. $e_8(0 \cdots 7) \leftarrow e_0 \cdot c_0(0 \cdots 7) + e_1 \cdot c_1(0 \cdots 7)$
$+e_2 \cdot c_2(0 \cdots 7) + e_3 \cdot c_3(0 \cdots 7) + e_4 \cdot c_4(0 \cdots 7)$
$+e_5 \cdot c_5(0 \cdots 7) + e_6 \cdot c_6(0 \cdots 7) + e_7 \cdot c_7(0 \cdots 7)$
$e_9 \leftarrow e_0 + e_1 + e_2 + e_3 + e_4 + e_5 + e_6 + e_7$
$e_{10}(0 \cdots 7) \leftarrow e_9 \cdot e_8(0 \cdots 7) + \overline{e_9} \cdot e_8(0 \cdots 7)$
$Gray\_level_{t_3}(0 \cdots 7) \leftarrow e_{10}(0 \cdots 7)$
Where $Gray\_level_{t_3}$ is the contone value of the template t_3.

Figure 5.7: Circuit after $s - LUTs$ to output contone value of template t_3.

5.1.9 Data-Path

A data-path implementing the described circuit is shown in Figure 5.8. It consists of all blocks that are described in the previous section. The data path is divided into rows, the first row contains registers that stores the templates fetched from the halftone image. The second row contains XM functions and third row contains registers that store results from XM functions. The fourth row contains "Demultiplexers" that have XM function values on select lines and they send the templates to "Multiplexer" adjacent to corresponding $s - LUT$. The fifth row contains "Multiplexers" that takes input from demultiplexers and output to the $s - LUT$ to which it is connected. The second last row contains eight smaller Look-Up Tables ($s - LUT$) because $N = 8$. The last row contains contone value copying circuit that is represented using Boolean equations in *Block 8* of the previous section.

5.1.10 Synthesis Results

Computational part of the proposed algorithms i.e., circuit without smaller Look-Up Tables ($s-LUT$) is implemented using VHDL (Very High speed integrated circuit Description Language) and the target platform in Xilinx Spartan 3E (xc3s100e-5vq100) FPGA. The design statistics obtained from Xilinx "Place and Route" tools show that it is small enough to be fit on a single Spartan 3E FPGA that has $< 100K$ gates. The synthesizes results are tabulated in Table 5.1, in which first column shows the FPGA slices that are consumed, second column shows maximum frequency the design can achieve on target FPGA, third column indicates the equivalent gate count of the

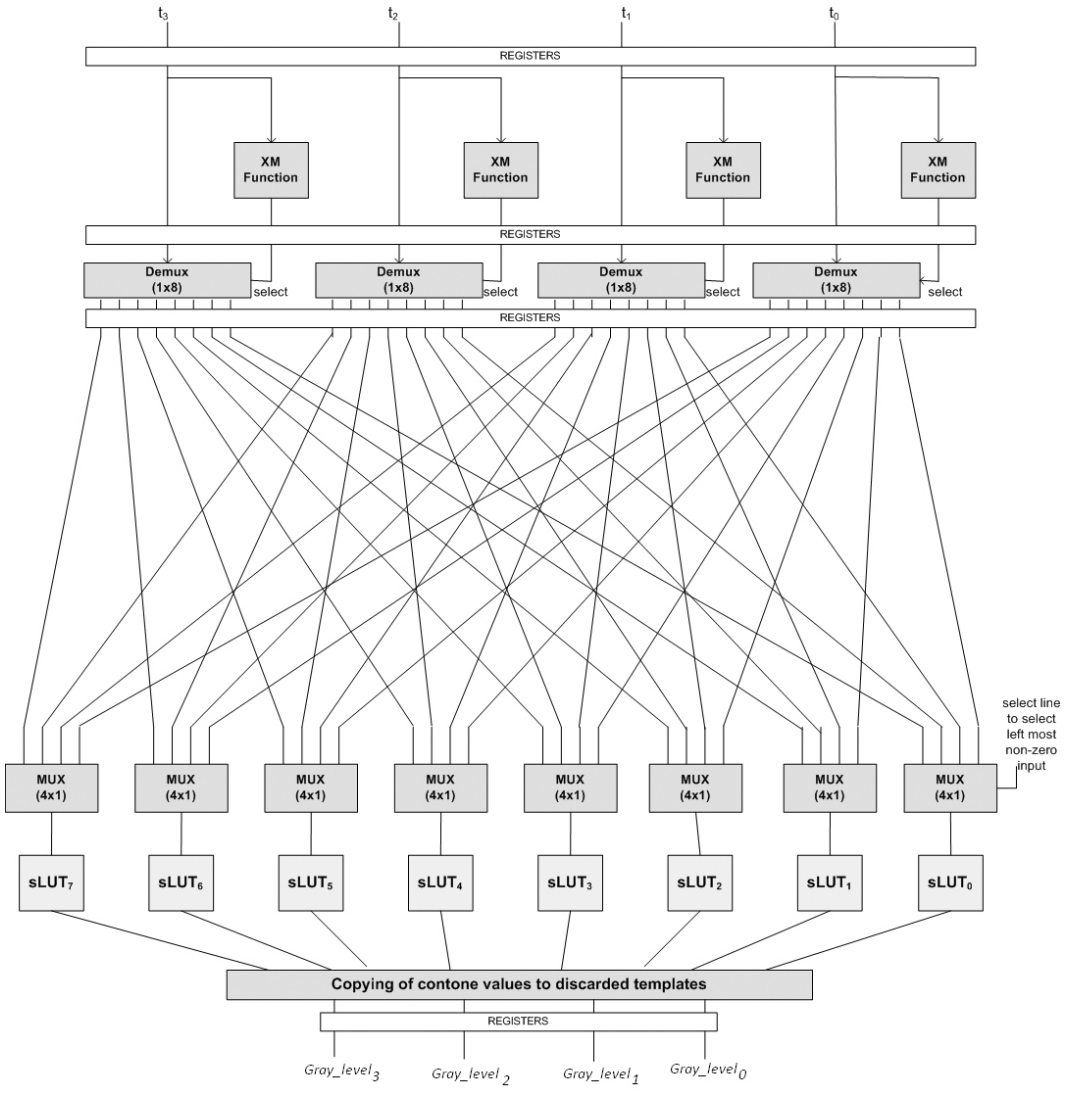

Figure 5.8: Data path of the circuit to implement XM_PL algorithm.

FPGA occupied area, and last column shows the power consumed by the design on target FPGA.

Table 5.1: Synthesis results of XM_PL algorithm.

Occupied Slices	Maximum Frequency	Gate Count	Power
958 out of 960	101.68 MHz	14,729	34 mW

5.2 Xor with Mean with No Pixel Loss (XM_NPL)

The integrated circuit that implements XM_NPL algorithm *(k=4 and N=8)* is similar to the integrated circuit of XM_PL algorithm except that it has the following changes: Block 6 is divided into two concurrent stages: Block 6a and Block 6b, these are also the last blocks in the integrated circuit.

5.2.1 Block 6a

This block consists of eight 8x1 multiplexers. The Boolean equations that represent the logic in this block in terms of previously computed variables are shown below:

$g_i(0 \cdots p+2) \leftarrow (sel_i(2) \cdot \overline{sel_i(1)} \cdot \overline{sel_i(0)}) \cdots A_3[i](0 \cdots p+2) + (\overline{sel_i(2)} \cdot sel_i(1) \cdot sel_i(0)) \cdot A_2[i](0 \cdots p+2) + (\overline{sel_i(2)} \cdot sel_i(1) \cdot \overline{sel_i(0)}) \cdot A_1[i](0 \cdots p+2) + (\overline{sel_i(2)} \cdot \overline{sel_i(1)} \cdot sel_i(0)) \cdot (A_0[i](0 \cdots p+2)$

where i=0 to 7 and i=0 refers to first multiplexer, i=1 refers to second multiplexer, and so on. The output g_0 is from the first multiplexer, g_1 is from the second multiplexer, and so on.

5.2.2 Block 6b

This block contains 8x4 $p+2$-bit registers such that 4 registers exist before each multiplexer. 4 registers store 4 inputs coming from demultiplexers to the corresponding multiplexer. This block also contains 8 identical Finite State Machines ($FSMs$) such that each FSM is connected to a distinct multiplexer. The FSM is shown in Figure 5.9 and consists of five states. It takes input from k registers connected to the corresponding multiplexer and its output goes to the *select line* of the same multiplexer. The Boolean equations representing operations performed in this block are shown below:

$sel_i(0 \cdots 3) \leftarrow \ FSM_i(A_3[i], A_2[i], A_1[i], A_0[i])$

where i=0, 1, 2, 3, 4, 5, 6, and 7.

In FSM, *Out* signal is connected to sel_i (select line) of the corresponding multiplexer and sequence number of template from first multiplexer is represented by sq_0, sequence number of template coming from second multiplexer is represented by sq_1, and so on. The FSM also has a signal $FINISH$ that is high when the FSM is supplying values to the multiplexer and becomes low when FSM comes back to its idle state. All pipeline registers maintain their last values when any $FINISH$ signal is high.

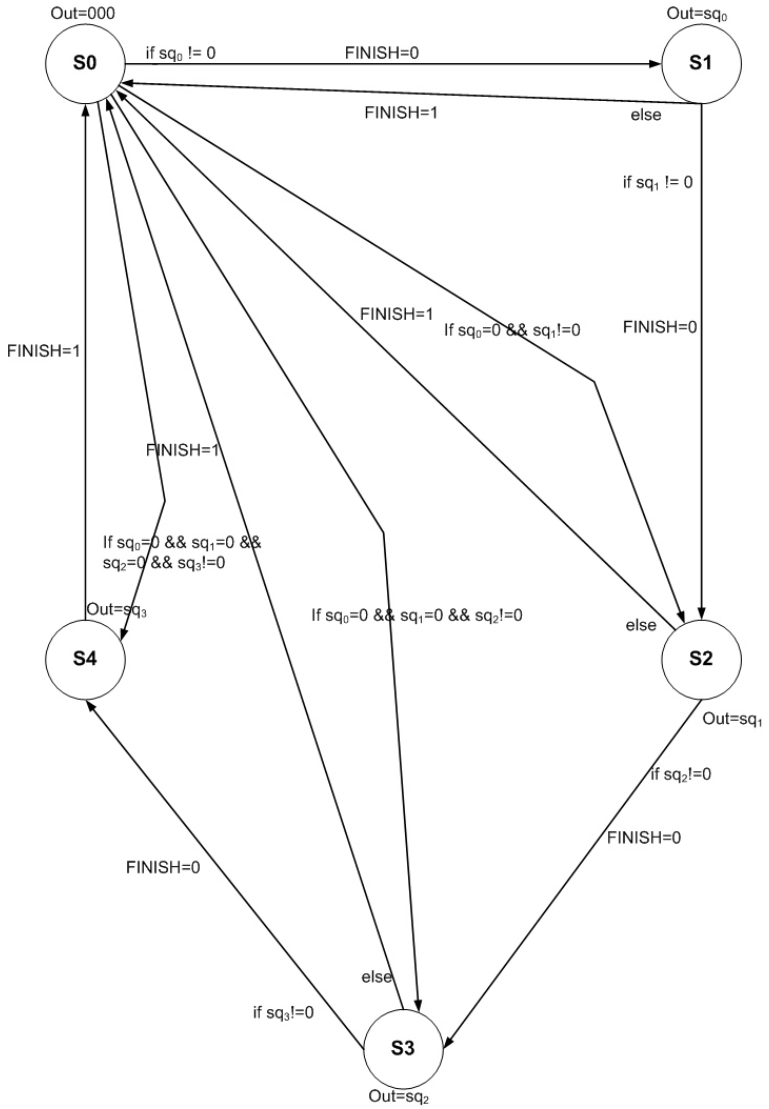

Figure 5.9: State diagram showing the control logic for select lines of multiplexers.

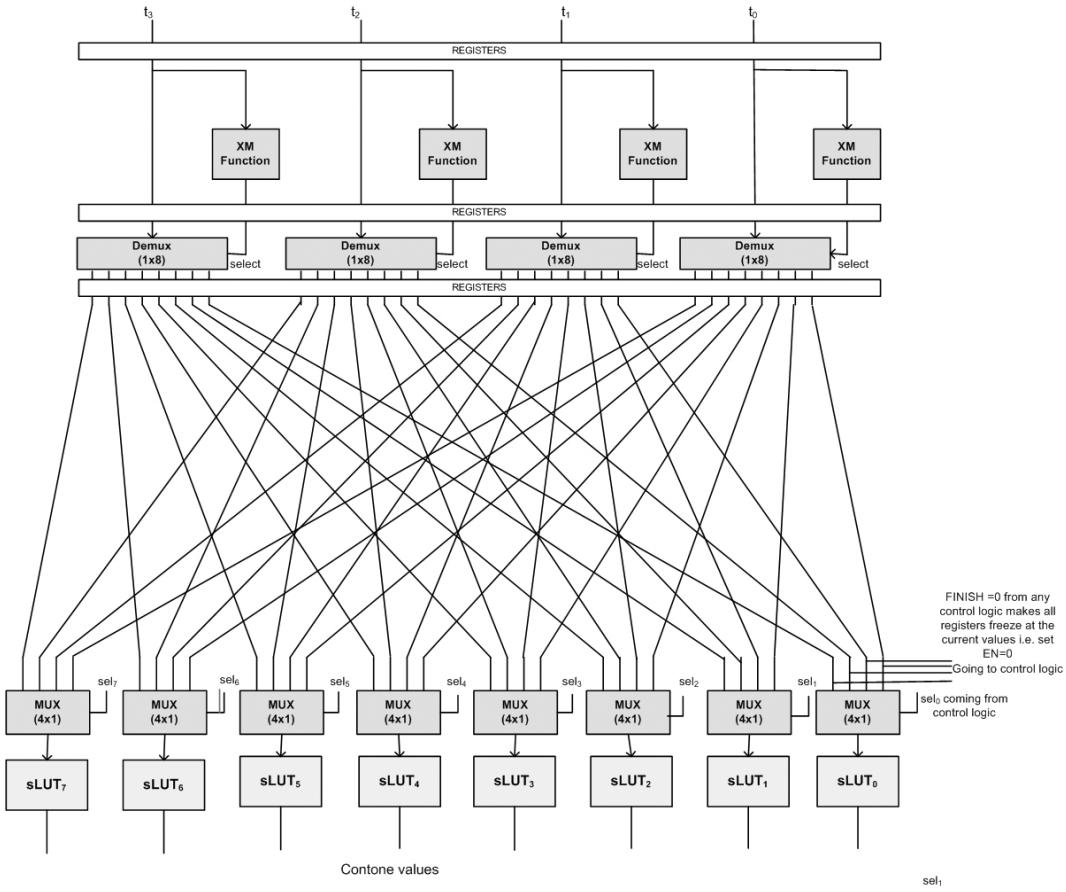

Figure 5.10: Data-path implementing XM_NPL algorithm.

5.2.3 Data-Path

The data-path to implement XM_NPL is similar to the data-path of XM_PL algorithm except that it does not contains *contone value copying circuit* and has $FSMs$ supplying inputs to the select lines of multiplexers that are present before $s - LUTs$. It is shown in Figure 5.10.

5.2.4 Synthesis Results

The VHDL model implementing computational part of the algorithm is synthesized for Xilinx Spartan 3E (xc3s100e-5vq100) FPGA. The results thus obtained are shown

in Table 5.2.

Table 5.2: Synthesis results of XM_NPL algorithm.

Occupied Slices	Maximum Frequency	Gate Count	Power
541 out of 960	112.97 MHz	9,161	34 mW

5.3 Only Addition with Pixel Loss (OA_PL) Algorithm

The integrated circuit to implement OA_PL algorithm with parameters k=4 and N=8 is similar to XM_PL algorithm except that *Block 3*, in which XOR operation is applied, is not present and outputs from *Block 2* go directly to *Block 4*. The total number of *Blocks* becomes equal to seven.

5.3.1 Data-Path

Data-path to implement OA_PL algorithm is described in Figure 5.11. It is similar to the data-path of XM_PL algorithm. The inputs are templates t_0, t_1, t_2, and t_3 and outputs are corresponding gray level values represented by $Gray\_level_0$, $Gray\_level_1$, $Gray\_level_2$,and $Gray\_level_3$.

5.3.2 Synthesis Results

Computational part of the OA_PL algorithm i.e., without $s - LUTs$ is synthesized for Xilinx Spartan 3E FPGA and the results are tabulated in Table 5.3.

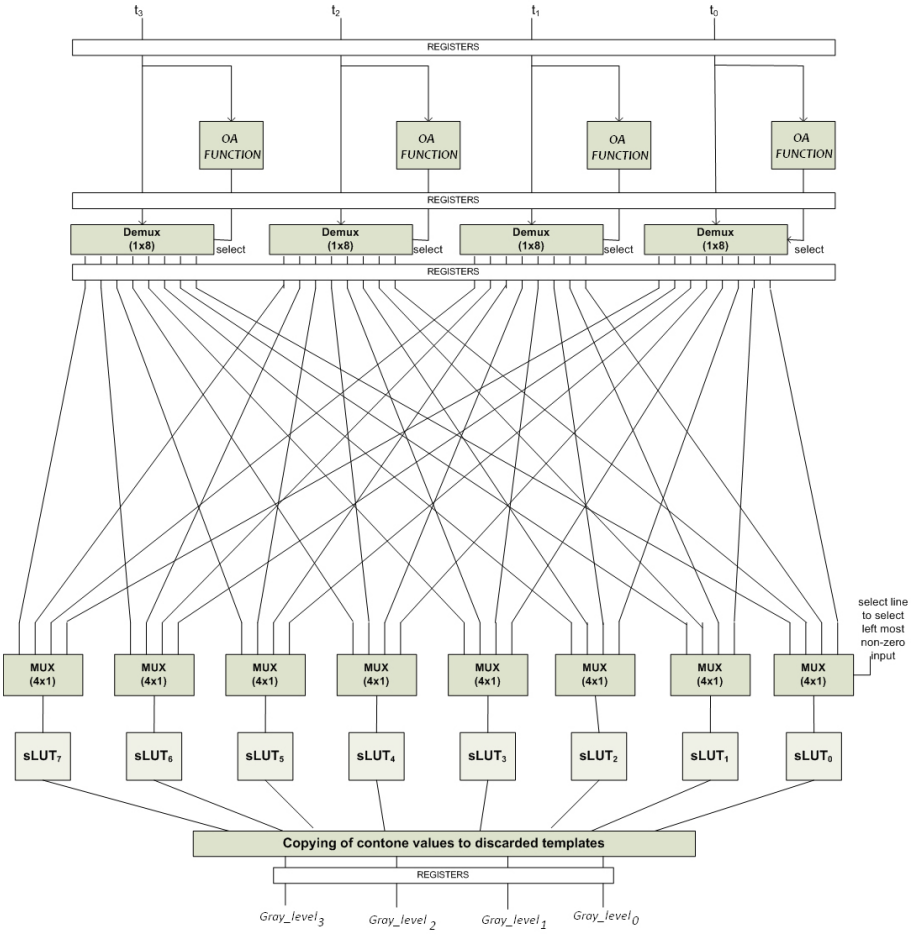

Figure 5.11: Data path to implement OA_PL algorithm.

Table 5.3: Synthesis results of OA_PL algorithm.

Occupied Slices	Maximum Frequency	Gate Count	Power
958 out of 960	112.394 MHz	14,207	34 mW

5.4 OA with No Pixel Loss (OA_NPL) Algorithm

This integrated circuit is similar to the integrated circuit of XA_NPL algorithm except that *Block 3* is not present and outputs from *Block 2* go to *Block 4* directly. Therefore it also has one *Block* less than OA_NPL algorithm.

5.4.1 Data-Path

This data-path is similar to the data-path of XA_NPL algorithm except that *OA function* blocks are present in place of *XM function* blocks. It is shown in Figure 5.12.

5.4.2 Synthesis Results

The synthesis of the computational part of the circuit is performed on Xilinx Spartan 3E (xc3s100e-5vq100) and the results obtained are tabulated in Table 5.4.

Table 5.4: Synthesis results of OA_NPL algorithm.

Occupied Slices	Maximum Frequency	Gate Count	Power
938 out of 960	112.978 MHz	9,158	34 mW

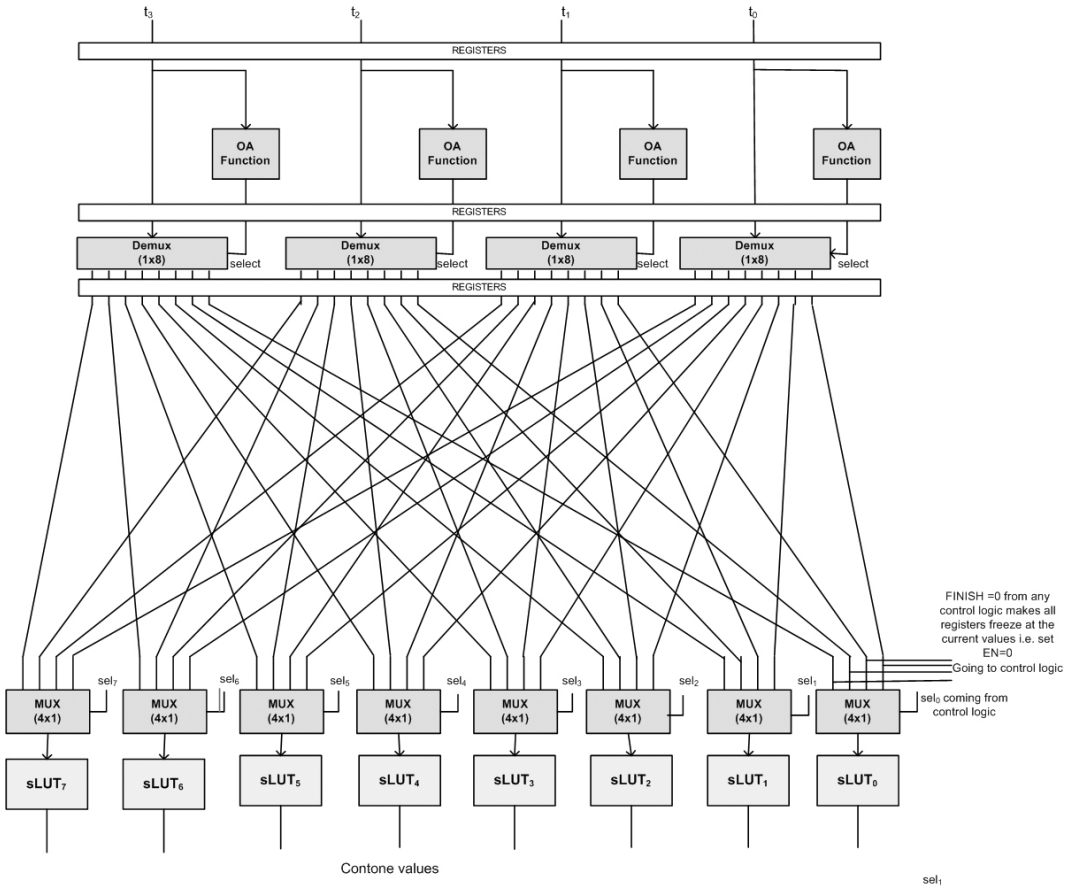

Figure 5.12: Data-path implementing OA_NPL algorithm.

5.5 Mod Only with Pixel Loss (MO_PL) Algorithm

The data path and hardware implementation of this algorithm is similar to XM_PL and OA_PL algorithms except that function MO is applied in place of functions XM or OA. Function MO is independent from any computation and only least significant $3 - bits$ ($log_2 N$) are kept to give the function result. The synthesis of MO_PL algorithm with parameters ($k = 4$ and $N = 8$) is performed using Xilinx ISE tools and results thus obtained are shown in Table 5.5.

Table 5.5: Synthesis results of MO_PL algorithm.

Occupied Slices	Maximum Frequency	Gate Count	Power
925 out of 960	112.978 MHz	13,160	34 mW

5.6 Mod Only with No Pixel Loss (MO_NPL) Algorithm

The data-path and hardware implementation of MO_NPL algorithm is similar to XM_NPL and OA_NPL algorithms except that function MO is used in place of function XM or OA. The synthesis is performed using Xilinx ISE tools and results thus obtained are shown in Table 5.6.

Table 5.6: Synthesis results of MO_NPL algorithm.

Occupied Slices	Maximum Frequency	Gate Count	Power
802 out of 960	164.553 MHz	12,120	34 mW

5.7 Discussion

From implementation results it is shown that proposed algorithms can be implemented on a FPGA. The designs can be fully pipelined with high throughputs i.e., clock frequency $> 100\ MHz$. Comparison of implementation results of both XM and OA algorithms recommend to implement algorithms based on simulation parameters: Image quality and gain in clock cycles over the serial LUT method. The reason is that both OA and XM algorithms do not differ significantly in occupied hardware area. The occupied area for PL algorithms is higher than NPL algorithms however, PL and NPL differ a lot in image quality and gain in clock cycles. Therefore in that case choice should be made based on image quality and speed up.

5.8 Summary

In this chapter we showed hardware implementations of the proposed four algorithms with *n=4 and N=8*. The target platform is Xilinx Spartan 3E FPGA with $100K$ gates, and Xilinx ISE 8.1i tools for VHDL are used to obtain results. The implementations of XM and and OA based algorithms take same amount of space. However, NPL (No Pixel Loss) algorithms save space over the PL (Pixel Loss) algorithms. This is due to replacement of *contone value copying circuit* with four $20 - bit$ registers ($p = 20$) in NPL algorithms. In *Discussions section* it is recommended to implement algorithms based on parameters image quality and the gain in clock cycles over the serial LUT method.

CHAPTER 6

CONCLUSION

Six new algorithms namely: Xor with Mean with Pixel Loss (XM_PL), Xor with Mean with No Pixel Loss (XM_NPL), Only Addition with Pixel Loss (OA_PL), Only Addition with No Pixel Loss (OA_NPL), Mod Only with Pixel Loss (MO_PL), and Mod Only with No Pixel Loss (MO_NPL) are proposed to perform parallel LUT (Look-Up Table) inverse halftoning. The algorithms can concurrently inverse halftone k pixels that are fetched from the halftone image. XM_PL, OA_PL and MO_PL algorithms offer almost $k\%$ gain in clock cycles over the serial LUT method whereas the image quality is slightly less than the serial LUT method. XM_NPL, OA_NPL, and MO_NPL algorithms have image quality same as serial LUT method and have $< 100\%$ gain in clock cycles over the serial LUT method. All algorithms can be implemented in terms of high throughput pipelined architectures. A single FPGA of $< 100K$ gates with external Content Addressable Memories (CAM) and Read Only Memories (ROM) can be used to completely implement an algorithm. It is also shown that any algorithm can be selected for hardware implementation based on its image

quality and the gain in clock cycles because they all consume almost similar hardware area and clock frequency.

REFERENCES

[1] Murat Mese and P. P. Vaidyanathan, *"Recent Advances in Digital Halftoning and Inverse Halftoning Method"*. IEEE Trans. Circuits and Systems I, June 2002.

[2] Ping Wong and Nasir D. Memon, *"Image Processing for Halftoning"*, IEEE Signal Processing Magazine, vol. 20, July 2003.

[3] Maire D. Reavy, Charles G. Boncelet, *"An Algorithm for Compression of Bi-Level Images"*, IEEE Tran. Image Processing, Vol. 10, No. 5, May 2001.

[4] A. N. Netravali and E. G. Bowen, *"Display of Dithered Images,"* Proc. SID, vol. 22, pp. 185-190, 1981.

[5] M. Y. Ting and E. A. Riskin, *"Error-Diffused Image Compression using a Binary to Gray Scale Decoder and Predictive Pruned Tree Structured Vector Quantization,"* IEEE Trans. Image Proceeding, Vol. 3, pp. 854-858, 1994.

[6] Murat Mese and P. P. Vaidyanathan, *"Look-Up Table (LUT) Method for Inverse Halftoning,"* IEEE Trans. Image Processing, vol. 10, October 2001.

[7] Tsi-Yi Chao and Hsueh-Ming Hang, *"Inverse Halftoning of Scanned Images,"* International Conference on Image Processing 1995, vol. 3, pages 420-423.

[8] P. C. Cheng, C. S. Yu and T. H. Lee, *"Hybrid LMS-MMSE Inverse Halftoning Technique,"* IEEE Trans. Image Processing, vol. 10, January 2001.

[9] Kuo-Liang Chung and Shih-Tung Wu, *"Inverse Halftoning Algorithm using Edge-Based Look-Up Table Approach,"* IEEE Trans. Image Processing, vol. 14, Issue 10, oct. 2005, pp. 1583-1589.

[10] R. Floyd, L. Steinberg, *"An Adaptive Algorithm for Spatial Grey-Scale,"* Proc. SID, pp. 75-77, 1976.

[11] T. D. Kite, D. Venkata, B. L. Evans, and A. C. Bovik, *"A Fast, High-Quality Inverse Halftoning for Error Diffused Halftones"*, IEEE Trans. Image Processing, vol. 9, Issue 9, Sept. 2000.

[12] M. Analoui and J. P. Allebach, *"New results on reconstruction of continuous-tone from halftone,"* IEEE Intl. Conf. Accoust. Speech Signal Processing, vol. 3, pp. 313-316, 1992.

[13] S. Hein and A. Zakhor, *"Halftone to continuous-tone conversion of error diffusion coded images,"* IEEE Trans. Image Processing, vol. 4, pp. 208-216, 1995.

[14] Z. Fan, *"Retrieval of images from digital halftones,"* ISCAS, pp. 313-316, May 1992.

[15] P. W. Wong, *"Inverse halftoning and kernel estimation for error diffusion,"* IEEE Trans. Image Processing, vol. 4, No. 4, pp. 486-498, 1985.

[16] Z. Xoing, K. Ramchandran, and M. Orchard, *"Inverse halftoning using wavelets,"* Proc. of Intl. Conf. Image Processing, Lausanne, CH, pp. 569-572, 1996.

[17] J. Canny, *"A computational approach to edge detection,"* IEEE Trans. Pattern Anal. Mach. Intell., vol. PAMI 8, No. 11, pp. 679-698, Nov. 1986.

[18] R. Neelmani, R. Nowak, and R. Baraniak, *"WInHD: Wavelet based Inverse Halftoning via Deconvolution,"* submitted to IEEE Trans. Image Processing, October 2002.

[19] Z. Xiong, M. T. Orchard, and K. Ramchandran, *"Inverse Halftoning using Wavelets,"* IEEE Trans. Image Processing, Vol. 8, No. 10, October 1999.

[20] A. K. Jain, Fundamentals of Digital Image Processing, Englewood Cliffs, NJ, Prentice=Hall, 1989.

[21] A. K. Katsaggetos (Ed.), Digital Image Resoration, New York, Springer-Verlag, 1991.

[22] R. Neelami, H. Choi and R. G. Baraniuk, *"Wavelet based deconvolution using the optimally regularized inversion for ill-conditioned systems,"* Wavelet Applications in Signal and Image Processing VII, Proc. SPIE, vol. 3813, pp. 58-72, July 1999.

[23] RD. L. Lau, G. R. Arce and N. C. Gallagher, *"Green Noise Digital Halftoning,"* Proceedings of the IEEE, vol. 86, pp. 2424-2442, December 1998.

[24] R. Eschbach and K. Knox, *"Error Diffusion Algorithm with Edge Enhancement,"* Journal of Optical Society Am. A, vol. 8, No. 12, pp. 1844-1850, December 1991.

[25] http://www.ccse.kfupm.edu.sa/∼umair

VITAE

Umair Farooq Siddiqi was born on April 8, 1979 in Karachi, Pakistan. He obtained his Masters of Science (MS) degree is Computer Engineering from King Fahd University of Petroleum & Minerals, Dhahran, Saudi Arabia in May 2007. He also have Bachelors of Engineering (BE) degree in Electrical Engineering from NED University of Engineering & Technology, Karachi, Pakistan since April 2002.

His areas of research include:

1. Algorithms and Computation,
2. Optimzation,
3. Parallel Computation, and
4. VLSI Design.

Umair Farooq Siddiqi can be contacted at `ufarooq.geo@yahoo.com`.